Titre: Title:	Picasso : un outil de co-design matériel/logiciel pour la conception de systèmes embarqués
Auteur: Author:	Yannick Héneault
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Héneault, Y. (2001). Picasso : un outil de co-design matériel/logiciel pour la conception de systèmes embarqués [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6955/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6955/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTRÉAL

PICASSO : UN OUTIL DE CO-DESIGN MATÉRIEL/LOGICIEL POUR LA CONCEPTION DE SYSTÈMES EMBARQUÉS

YANNICK HÉNEAULT
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MARS 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

PICASSO : UN OUTIL DE CO-DESIGN
MATÉRIEL/LOGICIEL POUR LA CONCEPTION
DE SYSTÈMES EMBARQUÉS

présenté par: HÉNEAULT Yannick
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. SAVARIA Yvon, Ph.D., président
M. BOIS Guy, Ph.D., membre et directeur de recherche
M. ABOULHAMID Mostapha, Ph.D., membre et codirecteur de recherche
M. GRANGER Louis, M. Sc., membre

À mes parents

Remerciements

Je tiens à remercier mon directeur de recherche, M. Guy Bois, ainsi que mon codirecteur, M. Mostapha Aboulhamid pour leurs bons conseils ainsi que leur soutien moral tout au long de mes recherches. J'aimerais également remercier M. Jacques Baillargé de Mentor Graphics qui, grâce à son aide technique, a été d'un soutien indispensable. Enfin, j'aimerais remercier le CRSNG et le GRIAO pour leur contribution financière pendant mes études.

Résumé

L'utilisation de logiciel et de matériel pour la conception de systèmes embarqués est une pratique de plus en plus courante. En effet, cela permet d'obtenir un meilleur rapport coût/performance. Pour réaliser ce type de conception, le système doit être décrit à un niveau d'abstraction très élevé. De plus, pour permettre de réduire le temps de conception, une méthodologie d'implantation rapide des mécanismes de communication entre les modules matériels et/ou logiciels doit être mise en place. On doit pouvoir aussi séparer les communications de la description comportementale des modules, pour permettre une réutilisation possible des différentes parties du système. De ces contraintes est apparu un nouveau concept : la synthèse des communications.

Ce projet a été accompli au sein du groupe *CIRCUS (*Système unifié de co-design par raffinement d'interfaces complexes. De l'anglais *Complex Interface Refinement for Codesign Unified System*) du GRIAO (Groupe de Recherche Interuniversitaire en Architecture des Ordinateurs). Ce projet de maîtrise a permis la mise au point d'un outil permettant de faire la conception de systèmes embarqués à un niveau d'abstraction élevé. L'objectif général est de développer un outil intégrant une interface graphique (CAO) permettant la saisie de spécifications système de haut niveau, en langage C (pour le logiciel) et en langage VHDL (pour le matériel). Des protocoles de communication de haut niveau sont offerts à l'usager, pour intégrer au sein même des spécifications, des façons d'échanger des données. Cela permet à des blocs matériels et logiciels de communiquer facilement entre eux, sans distinction Finalement, l'outil permet de convertir ces spécifications au niveau RTL pour faire la simulation et la synthèse du système complet. La covérification du système est effectuée avec l'aide du logiciel Seamless™ de Mentor Graphics.

Une nouvelle sémantique est ajoutée au C et au VHDL pour créer des mécanismes de communication entre ces deux langages. De plus, nous introduisons des algorithmes

de génération d'interfaces entièrement automatiques. Tous ces concepts ont été programmés dans un outil appelé Picasso. Certains de ces concepts sont nouveaux et font partie de la contribution de ce mémoire, alors que d'autres ont été pris d'outils existants et adaptés à notre méthodologie.

Ce projet s'inspire à la base de l'outil Renoir™ de Mentor Graphics, qui permet la saisie de spécifications matérielles uniquement. Afin d'étendre la fonctionnalité de Renoir et d'en faire un outil de codesign logiciel/matériel, notre approche considère également la saisie de spécifications logicielles et la synthèse d'interfaces de communication automatique. Le développement complet d'un outil de codesign constitue un projet complexe, qui ne peut évidemment pas être complètement réalisé dans le cadre d'un projet de maîtrise. Toutefois, ce projet jette les bases d'une plate-forme qui pourra par la suite être raffinée à l'intérieur d'autres projets de maîtrise. En ce sens, Picasso constitue donc une plate-forme de base pour d'éventuels projets de recherche.

D'un point de vue résultat, il sera démontré que la méthode implantée derrière l'outil Picasso permet de réduire de beaucoup le temps de conception d'un système embarqué (temps de mise en marché, communément appelé *time to market*). Nous verrons qu'elle permet également de faire automatiquement l'exploration de différentes architectures et la synthèse d'interfaces. Finalement, comme mentionné précédemment, nous verrons comment Picasso peut servir de plate-forme pour expérimenter différentes interfaces de communication, ainsi que leur version après-synthèse. Pour l'instant, un seul mécanisme de communication est supporté, mais puisque Picasso rend abstraits ces mécanismes à un haut niveau, différents protocoles de communication pourront être disponibles à l'intérieur d'une bibliothèque. Le concepteur pourra donc choisir, dans cette bibliothèque, le protocole de communication le plus approprié dans tel ou tel cas. Ce concept de bibliothèque constitue un domaine de recherche de pointe.

Abstract

Usually, embedded systems combine software and hardware modules. This gives a better cost/performance ratio. To realize this type of system, the design must be described at a very high level of abstraction, called system level description. Also, to decrease the time of design, a fast methodology must be used to implement the communication mechanisms between the hardware modules and/or the software modules. There must be a separation between the behavioral description of the modules and their communication interfaces to easily allow their reuse. From all these constraints appears a new concept: communication synthesis.

This project has been accomplished for the CIRCUS group *(Complex Interface Refinement for Co-design Unified System)* inside the GRIAO (*Inter-University Center in Computer Architecture and VLSI*). This master project implements a new tool used to capture and design embedded systems at a very high level of abstraction. The main goal was to develop a tool integrating a graphical interface to easily capture specifications. Such specifications can be in the C language for the software part and in the VHDL language for the hardware part. Some high level communication protocols are offered to the user. They can be used directly inside the specifications to exchange data. This facilitates sending and receiving data between hardware and software blocks, or between two software blocks implemented by two different microprocessors. Finally, the tool can convert the specifications to RTL to simulate the system with Seamless™, from Mentor Graphics. If the system is correct, we could then synthesize it.

New semantics were added to the C and VHDL languages to create communication mechanisms between these two languages. Also, new algorithms were developed to automatically generate interfaces. These concepts were programmed in the tool named Picasso. Some concepts are new and constitute the contribution of this thesis, but others were taken from existing tools and were adapted to our methodology.

This project can be seen as an extension of Renoir™, another tool from Mentor Graphics. This tool allows the capture of hardware specifications only. To extend the functionality of Renoir, and to create a software/hardware codesign tool, our approach considers also the capture of software specification and the synthesis of communication mechanism. The full development of a codesign tool is in fact a complex project that goes beyond the requirement of a master degree. However, this project constitutes the basis that could be refined later with other students doing master research. Thus, Picasso is a platform for other research projects.

It will be shown that the methods behind Picasso can reduce the time of design of an embedded system, and consequently the time to market. Also, it will be shown how Picasso can help to explore rapidly different architectures and to synthesize interfaces automatically. Finally, we will show how Picasso can be a platform to explore different kinds of communication interfaces and how they are applied. Indeed, at this moment, there is only one communication mechanism supported by Picasso, but since it is an abstract mechanism, different protocols can be added in the library. The designer can choose in the library what is the most appropriate mechanism depending on the context. This concept of libraries constitutes a very popular field of research.

Table des matières

DÉDICACE ..... **IV**

REMERCIEMENT ..... **V**

RÉSUMÉ ..... **VI**

ABSTRACT ..... **VIII**

TABLE DES MATIÈRES ..... **X**

LISTE DES FIGURES ..... **XII**

LISTE DES TABLEAUX ..... **XIII**

LISTE DES ANNEXES ..... **XIV**

LISTE DES SIGLES ET ABRÉVIATIONS ..... **XV**

AVANT-PROPOS ..... **XVI**

CHAPITRE 1 - INTRODUCTION **1**

1.1 PLAN DU TRAVAIL 6

CHAPITRE 2 REVUE DE LITTÉRATURE **8**

2.1 LA RÉUTILISATION 8

2.2 LES LANGAGES DE SPÉCIFICATION 11

2.3 EXEMPLE DE LANGAGE DE SPÉCIFICATIONS 12

2.4 LES ESTIMATEURS 14

2.5 LES OUTILS DE CO-DESIGN EXISTANTS 14

2.6 PROBLÉMATIQUE 17

2.7 RAFFINEMENT DE NOS OBJECTIFS 18

CHAPITRE 3 – L'INTERFACE ET LES SPÉCIFICATIONS DANS PICASSO **22**

3.1 LA MÉTHODOLOGIE DE PICASSO 22

3.2 CONCEPTS DE COMMUNICATION 27

3.3 LES MODIFICATIONS AUX LANGAGES 29

3.3.1 Langage de déclaration de types *29*

3.3.2 Extension au langage C *31*

3.3.3 Extension au langage VHDL *38*

3.4 SÉLECTION DES PROCESSEURS 41

CHAPITRE 4 – ARCHITECTURE CIBLE **42**

4.1 PRÉSENTATION GÉNÉRALE DE L'ARCHITECTURE DU SYSTÈME 42
4.2 LE MATÉRIEL DU BLOC *ICHIP* 47
4.3 LE LOGICIEL DU BLOC *ICHIP* 50
4.4 FONCTIONS SEND, INIT_THREAD, LOCK ET START_ENGINE 52
4.5 FONCTION « CHANGE_THREAD » ET LA MACRO PAUSE 54
4.6 FONCTION IO_POLLING 55
4.7 LE BLOC *ARCHI* 56

CHAPITRE 5 – ALGORITHMES DE TRANSFORMATION ET GÉNÉRATION 61
5.1 MISE À NIVEAU HIÉRARCHIQUE 62
5.2 REGROUPEMENT DU CODE LOGICIEL PAR INSTANCE DE PROCESSEUR 65
5.3 COMMUNICATION LOGICIELLE SUR UN MÊME PROCESSEUR 67
5.4 IMPLANTATION DES TYPES DE DONNÉES 68
5.5 RAFFINEMENT DES MODULES MATÉRIELS 69
5.6 GÉNÉRATION DES MODULES *ICHIP* 72
5.7 GÉNÉRATION DES DÉCODEURS DE REGISTRES DES *ARCHI* 75
5.8 GÉNÉRATION DU MODULE *ARCHI* 76

CHAPITRE 6 – APPLICATIONS PRATIQUES 79
6.1 LE FIFO 79
6.2 L'ADDITIONNEUR 87
6.3 LE RÉSEAU DE COMMUNICATION 89
6.4 ANALYSE DES RÉSULTATS 93

CHAPITRE 7 - CONCLUSION 94
7.1 TRAVAUX FUTURS 96

BIBLIOGRAPHIE 97

Liste des figures

Figure 1.1 Repartition logiciel matériel dans un SoC. Source [8] 4
Figure 2.1 Méthodologie de conception 20
Figure 3.1 Création d'objets 23
Figure 3.2 Instanciation de composants de la bibliothèque dans une fenêtre Picasso (approche ascendante) 24
Figure 3.3 Création de nouveaux objets (approche descendante) 25
Figure 3.4 Raffinement d'un bloc 26
Figure 3.5 Exemple de déclaration d'une structure 31
Figure 3.6 Client de l'additionneur en logiciel 32
Figure 3.7 Exemple d'utilisation d'une structure 35
Figure 3.8 Additionneur en logiciel 37
Figure 3.9 Additionneur en matériel 39
Figure 4.1 Architecture du système 43
Figure 4.2 Architecture à deux Ichips 47
Figure 4.3 Transformation du client logiciel par Picasso (avant & après). 52
Figure 4.4 Protocole pour la commande start pour une interface maître 59
Figure 5.1 Exemple d'une mise à niveau 64
Figure 5.2 Transformation de l'additionneur matériel par Picasso (avant & après) 71
Figure 5.3 Transformation du client logiciel par Picasso (avant & après) 73
Figure 6.1 Modélisation d'un FIFO en SystemC 80
Figure 6.2 Système utilisant le FIFO 81
Figure 6.3 Implantation du FIFO dans Picasso 83
Figure 6.4 Implantation du producteur (à gauche) et du consommateur (à droite) 84
Figure 6.5 Environnement Seamless 85
Figure 6.6 Traces de simulation du FIFO 86
Figure 6.7 Système à deux additionneurs 88
Figure 6.8 Le routeur 90
Figure 6.9 Structure de Tel1 90

Liste des tableaux

Tableau 4-1 Ports du Ichip 49
Tableau 4-2 Fonctions du Ichip 50
Tableau 4-3 Structure de données du Ichip 51
Tableau 4-4 Macros du Ichip 51
Tableau 6-1 Taille de l'exemple avant et après génération 88
Tableau 6-2 Signaux du client 91
Tableau 6-3 Comparaison du routeur avant et après génération 93

Liste des annexes

ANNEXE 1 - I_CHIP.H 100
ANNEXE 2 - DATA_TYPE.H 101
ANNEXE 3 - I960_1_CODE.C 102
ANNEXE 4 - I_CHIP.C 104
ANNEXE 5 - COPRO_TYPE_I960_1_CODE.VHD 106
ANNEXE 6 - ADDRESS_DECODE.VHD 109
ANNEXE 7 - ADDRESS_LATCH.VHD 111
ANNEXE 8 - BURST_LOGIC.VHD 112
ANNEXE 9 - BYTE_ENABLE_LATCH.VHD 114
ANNEXE 10 - SRAM_IF.VHD 116
ANNEXE 11 - TIMING_CONTROL.VHD 118
ANNEXE 12 - ARCH_TYPE_I960_1_CODE.VHD 121
ANNEXE 13 - COPRO_MASTER.VHD 122
ANNEXE 14 - COPRO_SLAVE.VHD 123
ANNEXE 15 - ARCHITEC_I960_1_CODE.VHD 124
ANNEXE 16 - ARCHI_I960_1_CODE.VHD 127
ANNEXE 17 - I_CHIP.VHD 130
ANNEXE 18 - CLIENT_V_FINAL_VHDL.VHD 138
ANNEXE 19 - ADDER_V_FINAL_VHDL.VHD 140
ANNEXE 20 - CLOCKER_FINAL_VHDL.VHD 141
ANNEXE 21 - PICASSO_TOP.VHD 142
ANNEXE 22 - DATA_TYPE.VHD 145

Liste des sigles et abréviations

API	Interface de programmation d'application. De l'anglais *Application Programming Interface*
Archi	Architecture de communication réduite. De l'anglais, *reduced communication architecture*
BC	*Behavioral compiler*
C	Le langage C
CAO	Conception assistée par ordinateur
CIRCUS	Système unifié de co-design par raffinement d'interfaces complexes. De l'anglais *Complex Interface Refinement for Codesign Unified System*
CRSNG	Conseil de recherches en sciences naturelles et en génie du Canada
FIFO	Premier arrivé, premier traité. De l'anglais *First-In First-Out*
GRIAO	Groupe interuniversitaire en architecture des ordinateurs et VLSI
Ichip	*interface for chip*
MFC	Classes de fondation de Microsoft. De l'anglais *Microsoft Foundation Class*
OCB	Bus sur une puce. De l'anglais, *On-Chip Bus*
RAM	Mémoire à accès aléatoire. De l'anglais *Random Access Memory*
ROM	Mémoire morte. De l'anglais *Read-Only Memory*
RPC	Appel de procédure à distance. De l'anglais, *Remote Procedure Call*
RTL	Logique à transfert de registres. De l'anglais *Register Transfer Level*
SLDL	Langage de conception au niveau système. De l'anglais *System Level Design Language.*
VC	Composant virtuel. De l'anglais *Virtual Component*
VHDL	Langage de description matériel des circuits intégrés allant à très haute vitesse. De l'anglais, *Very High Speed Integrated Circuit Hardware Description Language*
VSIA	*Virtual Socket Interface Alliance*
XDR	Standard de représentation de données externe. De l'anglais, *External Data Representation Standard*

Traductions francophones tirées du grand dictionnaire terminologique : www.grand-dictionnaire.com

Avant-propos

Puisque le groupe de recherche CIRCUS est tout nouveau, ce projet a pour objectif de devenir la pierre angulaire du groupe. C'est autour de ce programme que les étudiants viendront greffer leur projet, dans le but d'obtenir un ensemble unifié. L'architecture de Picasso est donc ouverte.

Si j'ai choisi le co-design, c'est qu'il s'agit d'un domaine de recherche de pointe pour lequel aucune solution commerciale n'est vraiment à la hauteur. Par contre, l'entreprise privée s'intéresse beaucoup aux solutions originales qui découlent du problème du co-design, ce qui en fait une activité très passionnante. Entre autres, notre partenaire Mentor Graphics suit de près le développement de mon outil.

Mon but est donc de produire un outil capable d'automatiser certaines facettes du co-design, plus précisément, celles qui touchent au domaine de la communication.

Chapitre 1 - Introduction

Dans cette section, nous verrons ce que sont les systèmes embarqués et ce qui les distingue des autres types de systèmes électroniques. Nous verrons leurs principales caractéristiques et comment ils sont décrits. Pour cela, nous aurons besoin de la notion de co-design logiciel et matériel. Nous verrons comment les systèmes embarqués ont évolué pour devenir aujourd'hui des *systèmes sur une puce*. Enfin, cette nouvelle intégration à grande échelle entraîne des problèmes de grande complexité, qui doivent être décrits par des langages puissants. Tout ceci nous permettra de présenter et de justifier nos objectifs généraux.

Les systèmes embarqués sont de plus en plus populaires dans le monde d'aujourd'hui. En télécommunication, on les retrouve sous la forme de téléphones cellulaires, de modems et autres composants réseaux. En imagerie, ils sont présents dans les cartes graphiques, les cartes d'acquisition, etc. On en retrouve également dans le domaine du contrôle et de l'automatisation comme dans les véhicules automobiles et les équipements industriels. La liste pourrait facilement s'allonger. Ces systèmes embarqués tirent leur nom du fait qu'ils réalisent un ensemble de tâches spécifiques. Les systèmes visent habituellement une cible spécifique. D'un autre côté, il faut prévoir une façon d'adapter le système à des changements dans la boucle de conception ou encore pour l'inclure dans des réutilisations futures (de l'ensemble ou de certaines parties). On doit donc concevoir le système en cherchant le meilleur compromis entre la spécialisation et la réutilisation.

Les systèmes embarqués sont également soumis à un ensemble de contraintes qualifiées de non-fonctionnelles [34]. Il y a tout d'abord le « coût de marge stricte ». Celui-ci indique, pour un intervalle de prix, la fonctionnalité attendue par d'éventuels consommateurs. Si le coût est supérieur à la plage acceptable, la fonctionnalité et les performances devront être à nouveau passées en revue.

On trouve également comme contrainte de conception le fameux « *time-to-market* » ou « temps pour la mise en marché ». Celui-ci est en général très court et, à la différence des autres systèmes électroniques, il doit tenir compte du temps de développement logiciel. Comme nous le verrons un peu plus loin, les systèmes embarqués possèdent en général des composants programmables (microprocesseurs). Le calcul du temps de mise en marché doit donc tenir compte du temps passé à créer l'électronique et ensuite du temps passé à la programmation. La troisième contrainte de conception appliquée aux systèmes embarqués temps réel est la présence de « contraintes de temps dures ». Il s'agit de contraintes de temps qui, lorsqu'elles ne sont pas respectées, peuvent entraîner de graves problèmes dans le système. Selon la tâche du système, il peut en aller de la sécurité et du bien-être des gens reliés aux systèmes.

La « consommation de puissance » est également un autre facteur à considérer. En effet, bien des systèmes embarqués sont compacts (téléphone cellulaire) et fonctionnent à piles. Il s'agit donc d'un facteur important à considérer. Enfin, on pourrait rajouter d'autres contraintes à la liste comme la sécurité ou les contraintes mécaniques comme le poids, les dimensions, etc.

Pour rencontrer ces contraintes et pour faire face à la complexité toujours croissante, plusieurs systèmes sont une composition mixte de logiciel et de matériel. L'architecture est donc formée d'une combinaison de noyaux de microprocesseurs programmables avec des mémoires et des périphériques matériels. Le choix de ce qui sera fait en matériel ou en logiciel s'appelle le partitionnement. Il est basé sur les critères suivants [28] : la performance du système, le coût de l'implantation, la réutilisation future, la transformation des données et la communication entre le logiciel et le matériel. Par exemple, on peut démontrer que le coût du développement du logiciel est plus bas que le matériel, que sa réutilisation et son entretien sont plus simples (changer la ROM par exemple), mais que l'exécution peut être ralentie par le manque de parallélisme.

Cette composition cause plusieurs défis aux concepteurs [5]. Déjà, par le fait que la conception du matériel et du logiciel est faite de façon très indépendante, ces parties sont généralement décrites dans des langages très différents, employant un formalisme distinct et des outils incompatibles entre eux! On n'a qu'à penser aux langages C et au VHDL. Le co-design logiciel/matériel est donc une tentative d'unir ces deux mondes par une méthodologie facilitant l'exploration de systèmes et d'architectures. Le principal avantage est de tenter de décrire le système dans sa globalité, c'est-à-dire avant même le partitionnement logiciel/matériel, i.e. le choix pour déterminer quelle (s) partie (s) de l'application sera implémentée en logiciel et quelle (s) partie (s) sera implémentée en matériel. Le co-design devient donc une méthode permettant de déterminer le meilleur équilibre entre le logiciel et le matériel qui respecte les contraintes précédemment énumérées relatives aux systèmes embarqués.

Si on regarde le contexte historique de l'évolution des systèmes, on remarque que le temps de mise en marché n'a cessé de diminuer. Les premiers langages abstraits permettaient de décrire un système électronique au niveau RTL. Puis sont apparus les compilateurs comportementaux permettant de transformer un algorithme, sous certaines conditions, en un code RTL. La même chose s'est produite pour les circuits imprimés sur plaquette (PCB). L'augmentation du nombre de transistors par puce et la vente de noyaux matériels (microprocesseurs, DSP, coprocesseur,...) a conduit à une nouvelle technique de conception appelée SoC (*system on chips)*. Cette façon de faire est similaire au circuit imprimé sur *PCB*. Il s'agit d'intégrer différents noyaux sur le même dé. Ces noyaux peuvent être vendus par diverses compagnies. L'avantage immédiat est le « temps de mise en marché », qui se trouve réduit par la réutilisation des composants [37].

La composition de ces SoC est majoritairement logicielle [8] (Figure 1.1). Une autre partie importante provient de la réutilisation de composants. Il ne reste donc qu'à bâtir la colle logique (en anglais, *glue logic*) qui servira à connecter le système de façon cohérente.

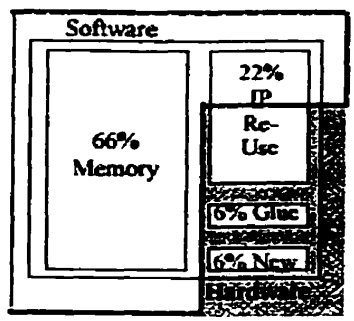

Figure 1.1 Repartition logiciel matériel dans un SoC. Source [8]

Pour gérer cette complexité, plusieurs outils ont été créés afin d'aider les concepteurs. Malheureusement, la vitesse de progression de ces outils est beaucoup plus lente que la croissance de la complexité des systèmes. Cet écart continue de s'agrandir et commence à causer certains problèmes quant à la capacité de conception [10][36].

La solution semble donc évidente, chercher à développer de nouveaux outils de co-design pour la mise au point de SoC. On peut distinguer deux types d'outils. Premièrement, les outils de conception de systèmes. Ceux-ci permettent d'entrer les spécifications dans un langage de haut niveau, puis de procéder par raffinements successifs, jusqu'à l'implantation finale. Le raffinement peut être complètement manuel ou assisté par des méthodes de conception automatisées. Le deuxième type d'outils sert à la simulation. Ils permettent de valider chaque raffinement. Alors que les outils de simulation sont aujourd'hui relativement bien établis (Seamless, EagleI, etc.) et donc de plus en plus utilisés, les outils de co-design n'en sont qu'à leurs débuts (Coware, VCC, etc.), et donc peu utilisés.

On se retrouve donc confronté au choix des langages dans les outils de conception système afin d'exprimer l'ensemble des contraintes et spécifications. Des tentatives de différentes natures sont apparues au cours des dernières années. Un langage pour la spécification au niveau système doit avant tout posséder une notation qui encapsule une sémantique pour décrire le système, avant que celui-ci ne soit relié à une architecture [35]. On doit donc établir la description sans penser aux types de microprocesseurs qui seront utilisés. Le langage doit aussi permettre l'entrée de contraintes pour permettre de vérifier si telle ou telle implantation est possible. Le langage doit également permettre une hiérarchie, afin de briser la complexité du système. Des niveaux d'abstraction différents doivent aussi être permis, afin de permettre plusieurs raffinements. Enfin un langage système doit permettre d'exprimer la notion de description de contrainte et de concurrence à différents niveaux de granularité.

En résumé, l'avenir des systèmes embarqués est grandement lié à la conception des SoC. Ceux-ci sont composés de plus en plus de blocs logiciels, mais une proportion doit quand même être implémentée en matériel, il doit donc y avoir un partitionnement des spécifications de haut niveau. Il y a également de plus en plus de blocs qui sont réutilisés. Le manque d'outils et de langage empêche le développement d'une méthodologie fiable et unifiée. La croissance marquante du nombre de transistors et la diminution du temps de mise en marché nous indiquent bien l'importance du problème. La construction d'un tel outil devra respecter la notion de langage système. La description de cette problématique nous permet donc de décrire nos objectifs généraux.

Ce projet a donc pour objectif l'élaboration et l'implémentation d'une méthode de conception niveau système. Le partitionnement étant un problème complexe en soi, nous allons supposer que celui-ci est spécifié. Il s'agit d'un partitionnement semi-automatique. Cela implique qu'on partitionne à la main, en décidant de ce qui sera en logiciel et en matériel. On peut quand même procéder à la synthèse des communications pour tester rapidement le système. La méthode va donc permettre de définir des blocs

logiciels ou matériels. La réutilisation sera supportée. La méthode permettra aussi la création du système, tout en respectant les exigences demandées. À savoir, placer les blocs logiciels sur des instances de processeurs et construire un système de communication permettant d'adresser les blocs matériels. Comme on a déjà expliqué qu'une description système se devait d'être abstraite, la communication entre les blocs devra être indépendante de leur nature logicielle ou matérielle. Donc, un bloc logiciel et un bloc matériel pourraient être interchangés s'ils ont la même fonctionnalité. Pour explorer différents partitionnements, on devrait donc avoir une représentation logicielle et matérielle de chaque objet partitionné. On peut donc au choix utiliser la version qui nous convient. La méthode synthétisera les communications correctement selon la nature des objets du système. Dans chaque cas, l'architecture finale serait évidemment changée, mais pas la fonctionnalité. Cette même fonctionnalité sera assurée par une co-simulation logicielle/matérielle.

Notre travail est donc une première approche dans la génération automatique d'architectures de système à partir d'une description dans laquelle le partitionnement est connu au niveau fonctionnel, mais sans le détail des communications. Étant donné la complexité du problème à résoudre, cette hypothèse simplificatrice demeure raisonnable dans le cadre d'un projet de maîtrise. D'autres projets traitant spécifiquement du problème de partitionnement automatique à l'aide d'estimateurs pourront éventuellement se greffer à ce projet,

1.1 Plan du travail

Le chapitre 2 présentera une revue de littérature visant à situer notre méthodologie dans l'univers du co-design. Au chapitre 3, on montrera comment les spécifications sont définies grâce à notre méthodologie. Le chapitre 4 présentera l'architecture choisie pour implanter ces spécifications. On y verra son fonctionnement, ses avantages et inconvénients. Le chapitre 5 expliquera quels sont les processus de génération qui

transforment les spécifications vers ce système final. Le chapitre 6 présentera quelques exemples qui ont été conçus dans Picasso. Enfin, le chapitre 7 présentera les travaux futurs et conclura le mémoire.

Chapitre 2 Revue de littérature

Les concepts généraux ayant été vus au chapitre 1, nous allons présenter quelques grands thèmes se rapportant au co-design. Dans chaque cas, les principaux outils ou concepts cités dans la littérature seront présentés. On traitera d'abord de la réutilisation. Puis, du concept de langage de spécification et de quelques exemples de langages relatifs à ceux-ci. Cela nous amènera à discuter du problème du partitionnement et de l'utilisation d'estimateurs. Enfin nous enchaînerons par une revue des principaux systèmes permettant de faire du co-design. En se basant sur cette rétrospective, nous terminerons par une présentation de nos objectifs spécifiques dans le cadre de ce mémoire.

2.1 La réutilisation

Pour favoriser et faciliter la réutilisation, le groupe VSIA *(Virtual Socket Interface Alliance* http://www.vsia.com) a produit un ensemble de règles pour aider à construire les systèmes SoC et pour faciliter l'échange et la vente de composants. Ce groupe fut formé en 1996 et rassemble plusieurs compagnies du monde de l'électronique. Son but est de pallier aux problèmes lors de la vente de composants virtuels. Par exemple, on y présente une façon de documenter un système pour adapter l'interface d'un composant à un nouvel environnement. VSIA tente, lorsque cela est possible, d'utiliser des normes déjà sur le marché, pour éviter la discrimination. Cela est très utile quand une norme est bien établie et peu dispendieuse.

Un composant virtuel est un composant qui peut être en logiciel ou en matériel. Il est virtuel dans le sens où il s'agit d'un composant qui est décrit dans un langage quelconque, au lieu d'être une pièce réelle. Le but est de formaliser la documentation accompagnant un composant pour faciliter son intégration dans un système. On tente de réduire le temps d'apprentissage en offrant un cadre qui englobe tous les détails

importants, selon une norme établie. On vise donc des principes communs de communication, de conception et de mesure d'assurance qualité [25].

Le moyen prôné par VSIA est de fournir un guide de conception d'interface. Ces interfaces permettent donc d'établir une distinction très claire entre le comportement d'un composant et son interface. Le composant devient donc une unité comportementale isolée du reste du système. Une interface peut être vue comme un ensemble de moyens différents pour envoyer et recevoir des données et des événements selon un certain protocole. On peut construire autant d'interfaces qu'il y a de protocoles. L'interface peut ensuite extraire les informations importantes et l'envoyer au composant. Ce qu'on appelle « informations importantes » est en fait une séquence d'événements et de données. Le composant modélise donc un « comportement », en fonction des stimuli à l'interface. Par exemple, on peut envoyer des données en utilisant un système de type requête/confirmation (*request/acknowledge*) ou encore, simplement utiliser une mise à zéro (*reset)* suivie d'un chargement automatique. Cela pourrait constituer deux protocoles différents, donc deux interfaces différentes.

On peut donc ainsi bâtir un ensemble d'interfaces en couche. On appelle ce modèle, le modèle en oignon, car à l'image de celui-ci, chaque interface représente une pelure et la fonctionnalité reste au centre. L'interface de plus haut niveau appelée « interface fonctionnelle » permet d'accomplir 4 opérations de base et elle est très restrictive. Voici ces opérations :

a) Les actions entre les ports ne peuvent être inter-reliées, donc avoir une dépendance (i.e. pas de *handshaking*).
b) N'importe quelle séquence sur ces ports est une propriété de comportement de haut niveau,
c) Les seules transactions valides sur les ports sont : la lecture, l'écriture, la sensibilité et l'émission. Les deux premiers étant relatifs aux données, les deux derniers utilisés pour les événements et le contrôle.

d) Les types de données qui passent à travers les interfaces peuvent être arbitrairement complexes.

L'interface fonctionnelle est notée 1.0. L'interface de plus bas niveau appelée interface d'implantation est appelée 0.0. Toute interface intermédiaire est donc notée 0.X, où X est un nombre entier.

Concernant VSIA, on retient donc deux conclusions importantes. Premièrement, la classification des protocoles et des propriétés de la communication dans une sémantique fixée. Cela permet une compréhension plus rapide des composants. Également, la distinction claire entre le comportement et l'interface ainsi que la décomposition en données et contrôle. Tout cela est dans le but d'augmenter le degré de réutilisation.

Une autre spécification du groupe VSIA est le modèle *OCB (*Bus sur une puce. De l'anglais, *On-Chip Bus*) pour les VCI (Composant virtuel à interface, de l'anglais *Virtual Component Interface*). Il traite de l'interconnexion de composants entre eux. Généralement, les composants d'un système vont échanger en utilisant un bus spécifique. Il est évident que VSIA ne peut prôner le choix de tel ou tel type de bus, pour des raisons évidentes de politique et d'économie. La solution est donc de décrire une encapsulation permettant de créer une interface au bus existant. Cette interface agira en convertisseur, permettant de connecter n'importe quel composant virtuel au bus. On appelle cette interface VCI.

L'avantage évident de VCI est l'interconnexion rapide pour la construction de plate-formes hétérogènes. Dans un système où différents processeurs cohabitent, si chaque processeur possède une interface VCI, ils peuvent être facilement raccordés entre eux.

2.2 Les langages de spécification

Il existe une multitude de langages de spécification. Chaque langage excelle dans des champs d'application bien précis. Le choix d'un langage est un compromis entre plusieurs critères comme le pouvoir d'expression du langage, la capacité d'automatisation des modèles décrits dans le langage et la disponibilité des outils supportant ce langage [19]. Dans les systèmes complexes, on retrouve donc bien souvent plusieurs langages différents décrivant les parties du système. C'est ce qu'on appelle un système hétérogène. À l'opposé, les systèmes homogènes sont entièrement décrits par un seul langage.

On retrouve dans la littérature quelques exemples de systèmes homogènes. Cosyma est un exemple [14]. Il sera abordé un peu plus loin. Polis [22] est un autre outil qui utilise Esterel comme langage d'entrée. Pour les systèmes hétérogènes, on utilise principalement les langages C et VHDL pour décrire le logiciel et le matériel. Coware [32] et Seamless [18] sont des exemples d'outils les utilisant.

Les langages de spécification système, comme on l'a vu au chapitre 1, doivent posséder certaines caractéristiques essentielles [29]. La notion de hiérarchie est la première caractéristique. Elle peut être décomposée soit d'une façon comportementale ou structurelle. Le niveau comportemental s'exprime en ordonnançant entre elles des tâches. Par exemple, lorsqu'une tâche est terminée, d'autres peuvent commencer. La hiérarchie structurelle, plus familière, exprime la décomposition d'instances en plusieurs sous-instances.

La seconde caractéristique est la concurrence. On peut réaliser cette caractéristique au niveau processus, gros grains, ou au niveau énoncé, petits grains. La troisième notion est la notion de synchronisme. Par celle-ci, on permet de fixer la durée d'une tâche, processus ou énoncé. Cela permet de guider vers un bon choix d'architecture

capable de respecter la spécification. Une autre caractéristique importante est la synchronisation. Il s'agit de décrire la transition entre 2 états, par exemple en utilisant un événement, un état ou une variable. D'autres caractéristiques qui, pour des raisons de simplification, ne feront pas l'objet de la discussion comme la communication inter-processus, les exceptions, les machines à états et l'interaction par rapport à la synthèse, occupent également une place importante dans la conception de systèmes embarqués.

2.3 Exemple de langage de spécifications

Cynapps [33] est un des premiers langages à modéliser un système électronique en utilisant des classes en C++. Certains efforts ont eu lieu avant, avec entre autre HardwareC [21] et HandelC [31]. Mais comme ces langages étaient basés sur le C uniquement, les notions d'instances d'objets, de synchronisation et de hiérarchisation sont vite devenues complexes à réaliser. L'avantage du C++ est qu'il permet, via le concept de classes, de modéliser la concurrence et la réactivité sans avoir à modifier le langage. En effet, il utilise la notion de classe au lieu d'ajouter de nouveaux mots réservés, ce qui rend le système portable pour différents compilateurs. La construction de classe permet donc d'ajouter au C++ les notions requises pour une description matérielle soit : la réactivité, la concurrence, l'instanciation hiérarchique et les types de données spécifiques. De plus, le C++ est connu par bien des concepteurs ce qui en fait un excellent choix.

La méthodologie de conception devient donc semblable à un raffinement itératif. Souvent la spécification première est donnée en C++. À ce niveau, elle est purement fonctionnelle. Aucune interface avec le matériel n'est précisée. Les concepteurs matériels doivent par la suite transformer cette description en Verilog ou VHDL pour construire le système matériel. Avec Cynapps, on peut poursuivre le raffinement avec la même description en intégrant progressivement les classes fournies. Cela amène l'introduction des caractéristiques du matériel précédemment nommées.

SystemC [8] est très semblable à Cynapps. Il contient également un ensemble de classes permettant d'intégrer au C++ les caractéristiques des systèmes matériels. Comme pour Cynapps, le raffinement se fait habituellement en 4 étapes [3] avec SystemC. La première étape décrit les spécifications sans la notion de temps. On l'appelle « sans cycle » (de l'anglais *untimed*). Le système est décomposé en modules qui communiquent par des canaux abstraits. La modélisation permet d'utiliser les processus élémentaires (*threads)* pour certaines parties. Les processus élémentaires permettent une exécution concurrente et efficace. La notion de temps est donc déficiente. On se sert de ce niveau pour évaluer la fonctionnalité du système.

Le deuxième niveau est appelé minuté. Il s'agit de fixer un délai d'exécution pour chaque processus de chaque module. Il ne faut pas confondre ce délai à celui d'une horloge. Il s'agit d'un délai fixé par l'usager qui représente la durée moyenne d'exécution du processus. Cela permet d'avoir une idée du temps d'exécution du système.

La troisième étape du raffinement consiste à passer au niveau bus (de l'anglais, *bus cycle*). À ce niveau, les modules sont synchronisés par un signal d'horloge. Certaines parties du système peuvent ne pas contenir d'informations relativement au temps d'exécution. Elles ne feront donc pas partie du calcul du temps d'exécution.

Au dernier niveau, le niveau cycle (de l'anglais *cycle accurate*), toutes les parties du système possèdent leur horloge. À ce niveau, le système est prêt pour la synthèse.

Il faut préciser que ces langages sont, à ce jour, orientés presque exclusivement pour décrire du matériel. L'aspect co-design n'est pas encore exploité par les concepteurs de ces langages. Selon l'échéancier établi par le groupe de développement de SystemC, cela devrait se concrétiser dans les prochaines années.

2.4 Les estimateurs

Très peu d'outils tentent de solutionner le problème du partitionnement de façon automatique. Cosyma [13] utilise un langage appelé Cx qui est un sur-ensemble du C. Des énoncés pour décrire les contraintes de temps et la définition de processus ont été ajoutés. L'algorithme de partitionnement est celui du recuit simulé. L'ensemble initial est une solution qui est entièrement en logiciel. La fonction de coût tient compte d'un ensemble de paramètres comme le temps d'exécution matériel (estimé) et logiciel (estimé), le délai de communication et le coût logiciel. L'algorithme peut donc déplacer des blocs logiciels en matériel et vice-versa, procéder à l'estimation et calculer la fonction de coût. Il raffine ensuite sa solution initiale dans l'espoir de trouver une meilleure solution par un meilleur partitionnement. Cette approche, par contre, sans de bonnes heuristiques, donne des résultats de qualité moyenne [14].

2.5 Les outils de co-design existants

Comme on l'a mentionné lors de l'introduction, un sous-problème du co-design est la co-simulation. Il faut être capable, à différents niveaux d'abstraction, de vérifier si les spécifications sont toujours bonnes. On parle de co-simulation lorsqu'une spécification logicielle est simulée conjointement avec une spécification matérielle. On utilise donc différents simulateurs et le co-simulateur s'occupe de la gestion de ceux-ci. L'outil de co-simulation commercial qui détient la plus grande part de marché dans cette catégorie est Seamless CVE de Mentor Graphics. Celui-ci permet de vérifier le comportement logiciel/matériel d'un système embarqué. Cela permet donc à l'équipe logicielle d'interagir plus tôt dans le cycle de développement du produit.

Seamless CVE possède ses propres modèles de microprocesseurs employés dans le système. Le modèle permet d'obtenir la bonne interaction des signaux pour le simulateur matériel, mais aussi de simuler les jeux d'instructions et leur implication pour

le simulateur logiciel. Il est aussi capable de synchroniser entre eux un simulateur matériel et un simulateur logiciel.

Au niveau des outils de saisie de spécifications, on retrouve naturellement Coware [32]. Coware est un environnement permettant la saisie de systèmes où on vise surtout la modularité [32]. Dans Coware, l'unité de base est le processus. Un processus peut-être codé en C, DFL, VHDL ou avec le « langage Coware». Cela implique que le partitionnement logiciel/matériel est entièrement décidé par l'usager selon le choix du langage. Un processus décrit un comportement. Il peut communiquer avec d'autres processus par l'intermédiaire de ports. Les ports sont reliés entre eux par un canal, ce qui permet d'établir la communication. Le paradigme de communication utilisé entre les deux processus est le RPC (appel de procédure à distance, de l'anglais *Remote Procedure Call*), emprunté à la programmation distribuée.

Le RPC consiste à un appel de fonction à distance. En programmation classique, lors de l'appel d'une fonction, on commence par lui passer des paramètres. Puis le contexte d'exécution passe au corps de la fonction. Enfin, la fonction retourne une valeur de retour et le programme continue. En ce qui concerne le RPC, le principe est le même, excepté que le client, celui qui appelle la fonction, et le serveur, celui qui représente la fonction, sont deux processus différents qui peuvent s'exécuter sur deux ordinateurs différents. Le client envoie donc les paramètres et reste en attente. Pendant ce temps, le serveur exécute la fonction et envoie un résultat à la fin. Cela débloque le client qui reprend son exécution en lisant la valeur de retour.

Le concept de RPC s'applique donc bien aux processus. On sait qu'un processus logiciel peut être modélisé par un processus élémentaire et qu'un processus matériel est modélisé par un « *process* » en VHDL. Ceci permet donc à Coware de se servir du RPC pour définir à un haut niveau une communication logicielle/logicielle, logicielle/matérielle, matérielle/matérielle. La communication inter-processus est donc

définie par les RPC uniquement. Lorsque plusieurs processus élémentaires se retrouvent dans le module, on parle alors de communication intra-processus. Celle-ci peut se faire en utilisant des variables partagées ou des signaux produit par les modules.

De cette description, il est possible de procéder à une simulation de haut niveau pour vérifier la fonctionnalité du système. À ce stade, on ne considère pas une architecture en particulier. On exécute tous les processus sur la machine hôte en simulant les appels RPC. Cela permet une simulation beaucoup plus rapide que le niveau porte par exemple.

Une fois la fonctionnalité validée, on peut associer les processus logiciels à un microprocesseur en particulier. Coware ne supporte que la synthèse logicielle pour des systèmes monoprocesseur. Puisque le partitionnement est déjà décidé, les modules en C peuvent être placés sur le microprocesseur. Un micro-noyau est ajusté pour permettre l'exécution des différents processus élémentaires. Pour faire le pont avec les modules matériels, les différents canaux sont raffinés, ce qu'on appelle la synthèse de communication. L'implantation de la communication peut-être faite par [27] mémoire partagée, par instruction assembleur spécifique ou par interruption.

En résumé, Coware offre une méthodologie permettant la réutilisation de composants dans un cadre très spécifique.

2.6 Problématique

Mentor Graphics, notre partenaire industriel, désire un outil de co-design. Notre mandat est donc de créer un prototype. Le seul outil semblable de Mentor s'appelle Renoir. Renoir permet la saisie de spécifications matérielles seulement. Il utilise pour cela des représentations graphiques standards. Par exemple, avec Renoir, on peut entrer graphiquement des machines à états, des diagrammes blocs et des tables de vérité. L'outil peut traduire ces représentations vers des descriptions matérielles (VHDL ou Verilog). Malheureusement, Renoir ne supporte pas le logiciel. L'outil ne génère donc qu'une représentation fidèle en VHDL ou Verilog de l'information entrée par l'interface usager. L'optimisation est laissée au compilateur. Renoir est donc un simple outil de traduction.

Pour ne pas réinventer la roue, les idées de Renoir seront reprises : interface usager et certains éditeurs. Le code ne nous est pas fourni, nous devrons donc faire notre prototype de zéro.

Pour la simulation de systèmes embarqués, nous avons vu que Mentor Graphics possède un outil appelé Seamless. Le problème avec Seamless est que, contrairement à Renoir, sa courbe d'apprentissage est très ardue. Pour simuler un système dans Seamless, il faut créer tout le circuit, paramétrer les mémoires et les processeurs utilisés, les simulateurs ainsi que d'autres facteurs. Seamless aurait donc besoin d'un autre outil permettant d'agir en façade (*front-end*) afin de simplifier cette tâche.

Notre prototype devra donc également s'acquitter de cette contrainte. Ainsi, il devra permettre l'entrée de spécifications en utilisant une interface graphique et des langages de haut niveau. De cette description, il devra générer le système final, prêt à être simulé avec Seamless.

2.7 Raffinement de nos objectifs

Comme on a mentionné au chapitre précédent, notre outil devra permettre de construire une représentation de haut niveau du système embarqué. Pour cela, on devra permettre l'utilisation de langages de haut niveau. Notre système sera hétérogène. Comme une des hypothèses était que le partitionnement est fixé à l'avance, il devient plus simple de considérer le langage C/C++ comme un langage de haut niveau pour les blocs logiciels, et le VHDL pour les blocs matériels. C'est donc ces deux langages qui seront retenus.

On devra offrir à l'usager des mécanismes de communication de haut niveau. Ces mécanismes serviront à échanger les données entre les blocs. Ils devront être indépendants de la nature du bloc. Comme on l'a dit, on pourrait changer un bloc logiciel pour un bloc matériel qui a la même fonctionnalité sans changer les communications. On peut faire cela car les communications sont définies de façon abstraite. On doit définir une façon de communiquer qui soit indépendante de chaque langage mais qui peut être supportée par ceux-ci. Les mécanismes seront supportés par des extensions aux langages.

La saisie des spécifications se fera à travers une interface graphique. Renoir sera le bon point de départ. Cela permettra de saisir facilement l'ensemble des spécifications et d'effectuer les tâches nécessaires, entre autres, l'établissement de liens de communication entre différents blocs ou entités. Pour chaque bloc, l'usager définira une interface de communication. Ensuite, en créant des instances de ces blocs, on pourra les connecter ensembles pour permettre la communication. Cela est l'approche envisagée pour la saisie du système. Une fois toutes ces informations entrées, l'outil devra être capable de décomposer ces interfaces en code C et VHDL standard pour simuler et réaliser le système final.

Pour permettre l'établissement de liens de communication, des interfaces seront conçues pour permettre une modélisation de haut niveau. Ces interfaces renferment les protocoles de communication pour un microprocesseur et permettent à des blocs matériels quelconques de se brancher sur un bloc logiciel selon un protocole déjà établi à l'avance. C'est l'idée de VSIA qui est reprise ici. Ces interfaces, appelées Ichip, seront présentes dans une bibliothèque qui pourrait être étendue au besoin à différents types de microprocesseurs. Cela a pour avantage de cacher l'aspect « communication » d'un système au concepteur, laissant à l'outil le travail de choisir et de réaliser correctement les communications.

La possibilité de pouvoir facilement explorer les différentes façons d'implanter un système constitue un aspect important pour les architectes de système. Ils doivent décider quelles parties du système se feront en logiciel et en matériel, et décider du type des microprocesseurs à utiliser, choisir le nombre de microprocesseurs et les mécanismes de communication. Voilà autant de points qui, malgré l'expérience des ingénieurs, peuvent changer en cours de route. On doit donc tenir compte de ce fait dans l'élaboration des spécifications du système et avoir un outil qui permet facilement de corriger ces éléments, soit par exemple en changeant le code de chaque microprocesseur, ou en créant automatiquement des bus de données entre les microprocesseurs.

La Figure 2.1 illustre la méthodologie de conception qui sera présentée pas à pas dans ce mémoire. La première étape consiste à préciser les spécifications du système en C et en VHDL. Pour cela on utilise l'interface usager pour saisir les objets C et VHDL. Pour chaque langage, des extensions ont été apportées pour permettre les communications. Ceci fera l'objet du chapitre 3.

Le système final est composé de microprocesseurs spécifiques exécutant le code C. La communication se fait par mémoire partagée. À ce niveau, les codes C et VHDL

deviennent donc des codes purs, c'est-à-dire sans les extensions utilisées dans la spécification. L'explication de cette architecture sera faite au chapitre 4.

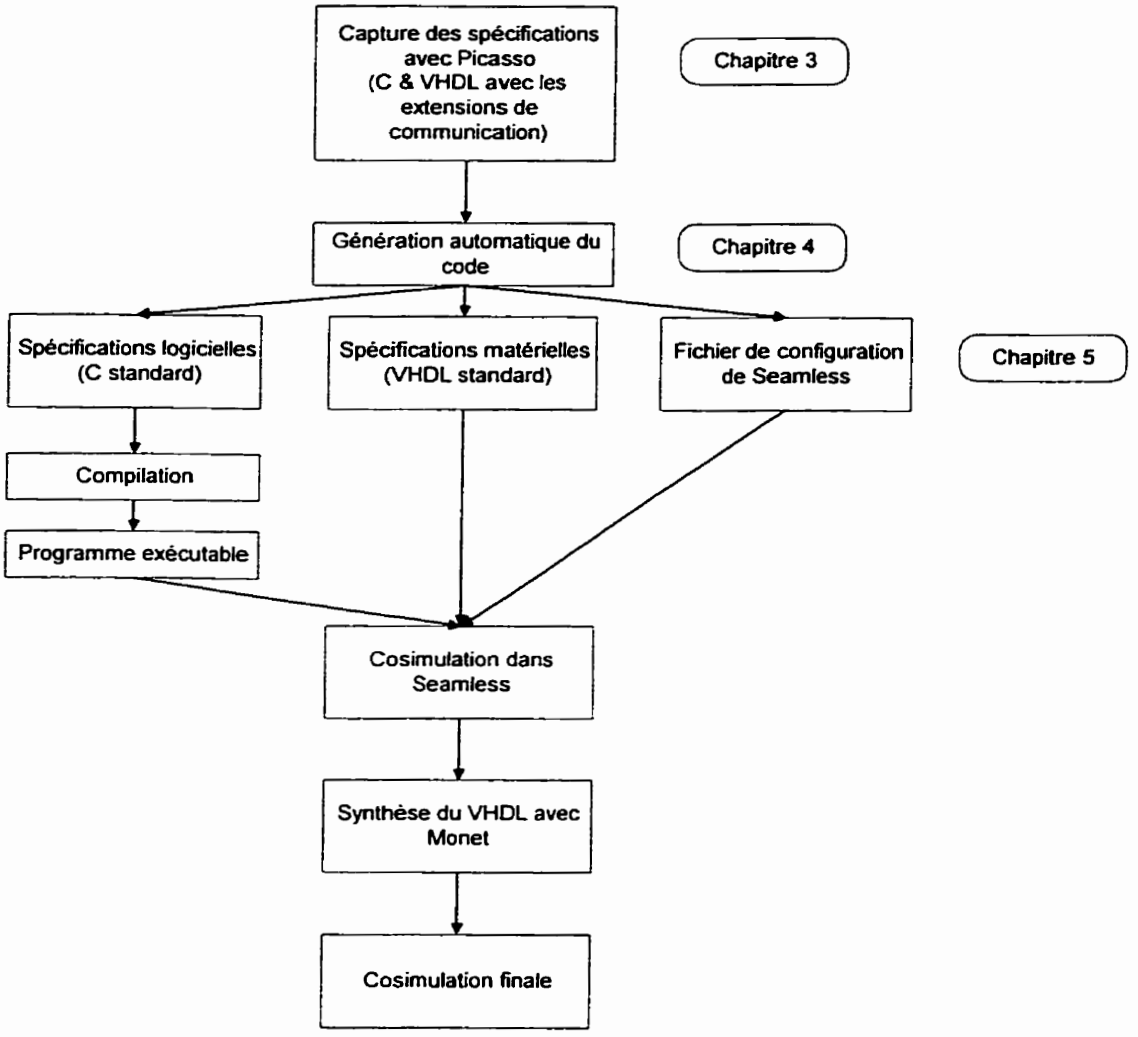

Figure 2.1 Méthodologie de conception

Pour passer des spécifications entrées à l'architecture du système, des algorithmes de génération doivent être utilisés. Ils seront vus au chapitre 5. À cette étape, on génère également les fichiers de configuration pour l'outil Seamless et un ensemble de *scripts* pour compiler le code C et VHDL. Cela permettra de vérifier, en simulation, si les

spécifications entrées et le raffinement fait par Picasso sont corrects. Si ce n'est pas le cas, on devra recommencer avec de nouvelles spécifications.

Si le système est valide, on peut alors synthétiser le code VHDL. Le code généré par Picasso est de niveau RTL, il peut donc être synthétisé sans problème. Tout dépend en fait de la spécification entrée par l'usager. En effet, le travail de Picasso se fait simplement au niveau de la communication. Picasso n'interfère en rien avec le code entré par l'usager. Donc si le code entré par l'usager est de niveau comportemental, alors un compilateur permettant de le synthétiser devra être utilisé. Cela fonctionnera même si les communications sont détaillées au niveau RTL.

Le chapitre suivant présentera un article reprenant la problématique de l'outil, la méthodologie de design ainsi qu'une synthèse de chaque étape.

Chapitre 3 – L'interface et les spécifications dans Picasso

Dans ce chapitre, nous allons introduire l'outil Picasso. Nous verrons plus en détails l'interface usager et les langages utilisés pour décrire un système embarqué. Cela permet de décrire le système et donc constitue la première étape de la méthodologie. L'interface de Picasso permet une approche de conception descendante et ascendante à la fois, ce qui simplifie l'entrée des spécifications. Puis, les différents paradigmes de communication seront abordés. On verra entre autres que des mécanismes de communication synchrones, comme les RPC, et asynchrones sont supportés. Cela sera vu en présentant la sémantique de communication.

Nous traiterons également du langage mis au point pour définir la notion de types abstraits. Ces types identifient les données qui passent sur un lien de communication. L'adjectif abstrait indique qu'ils sont indépendants de la nature logicielle ou matérielle de l'implantation. Enfin, nous compléterons par les extensions apportées aux langages déjà existants. Dans le cas du C, nous verrons le support fourni pour la communication et l'ajout de processus élémentaires. En VHDL, nous traiterons des descriptions mixtes. Celles-ci permettent d'inclure deux niveaux d'abstraction dans une même description VHDL.

3.1 La méthodologie de Picasso

Picasso utilise la notion de projet pour gérer la description d'un système. Un projet contient l'ensemble des spécifications et des liens de communication définis par l'usager. On y retrouve tout ce qui est nécessaire pour décrire complètement le système.

Deux approches sont permises pour construire un système : l'approche descendante et l'approche ascendante. La première innovation est donc d'offrir à la fois ces deux paradigmes de pensée aux concepteurs. La flexibilité de l'outil est accrue, étant

donné qu'on se rend compte en pratique que les concepteurs changent souvent d'approche pendant la mise au point d'un même système. En effet, il peut être difficile de tout concevoir le système selon une des deux approches, surtout au tout début de l'activité de conception.

Nous allons donc illustrer ces deux approches. Commençons par l'approche ascendante. Comme le montre la Figure 3.1, trois types d'objet peuvent être créés par Picasso, soit : le type Picasso, logiciel ou matériel. Les objets Picasso sont des objets offrant une structure hiérarchique contenant d'autres objets et des interconnexions. Quant aux objets logiciels et matériels, ils correspondent à du code en C ou en VHDL, incluant les extensions de communication.

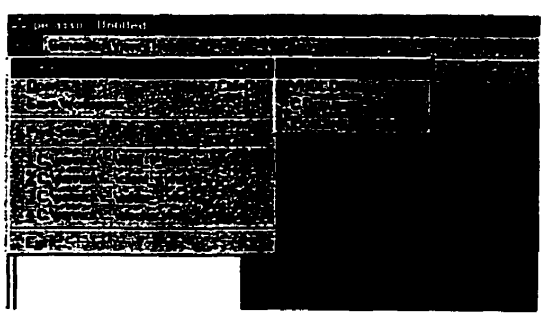

Figure 3.1 Création d'objets

Chaque objet dispose de son propre type de fenêtre d'édition. Cette fenêtre permet de spécifier l'objet. Les objets de type Picasso sont structurels et sont définis de façon graphique. On peut ainsi illustrer les instances accompagnées de leurs liens de communication. Les objets matériels sont plutôt spécifiés par un code VHDL. Quant aux objets logiciels, ceux-ci sont décrits en C. Des primitives de communication équivalentes sont ajoutées à chacun de ces langages. Ces objets deviennent disponibles par la suite dans la bibliothèque comme composant de base dans la construction d'autres objets du système.

Un exemple de fenêtre d'édition d'objet Picasso est illustré à la Figure 3.2, sous le nom de « *top* ». Un objet Picasso représente les instances qu'il contient sous forme de

boîtes. L'objet Picasso agit également comme un composant du projet. Il est réutilisable comme instance (boîte) dans d'autres objets Picasso. À la Figure 3.2, on suppose qu'un objet « adder » est dans la bibliothèque. Celui-ci est soit en logiciel ou en matériel. Puis, dans la fenêtre graphique « top », quelques instances de cet objet sont créées. Il s'agit donc d'un schème de construction ascendante. Plus précisément, chaque fois que nous montons d'un niveau, nous pouvons créer des instances provenant des niveaux inférieurs. Dans chaque niveau, il est possible d'établir des relations entre ces instances.

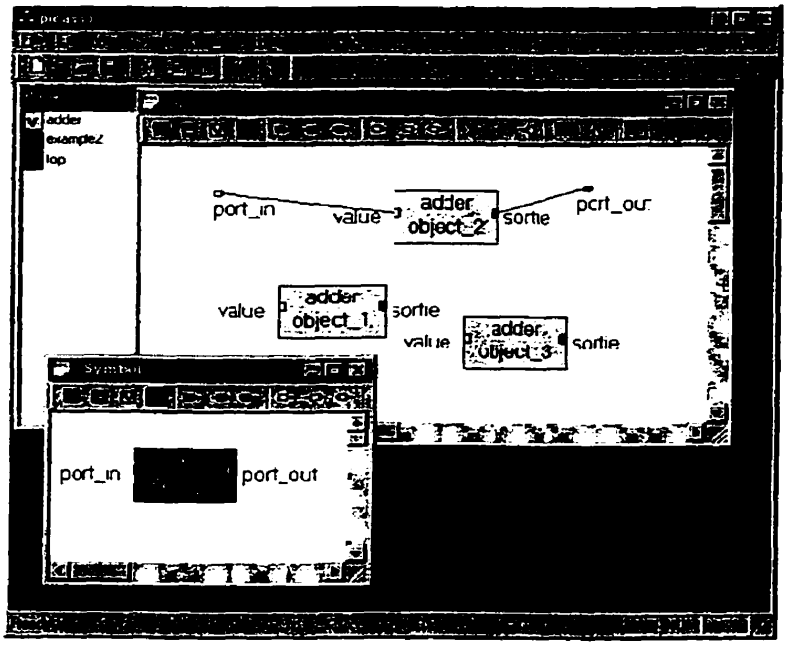

Figure 3.2 Instanciation de composants de la bibliothèque dans une fenêtre Picasso (approche ascendante)

Chaque objet communique par des ports. Il s'agit de points d'attache sur lesquels on peut établir des liens de communication. Ainsi, deux blocs peuvent communiquer en connectant par un lien (ligne) de communication un port de chacun des blocs. Puisqu'on est dans une approche ascendante, on doit disposer d'un mécanisme permettant de joindre les différents niveaux à mesure qu'on monte de niveau. Pour cela, Picasso permet de créer des ports « flottants ». Ainsi, l'ensemble de ces ports flottants définit l'interface de l'objet Picasso. Dans la Figure 3.2, la fenêtre « *symbol* » contient une instance de l'objet

« *top* ». Les deux ports flottants appelés « *port_in* » et « *port_out* » se retrouvent sur la boîte représentant l'instance.

Passons à l'étude du paradigme de création descendant. Chaque fenêtre d'objet Picasso offre la possibilité de créer nos trois types d'objet (Figure 3.3). Ainsi, on peut créer plusieurs nouvelles instances de types différents dans la fenêtre d'édition graphique. Sur ces nouvelles instances, des ports peuvent être créés. On définit donc l'interface d'un composant avant même de spécifier son implantation.

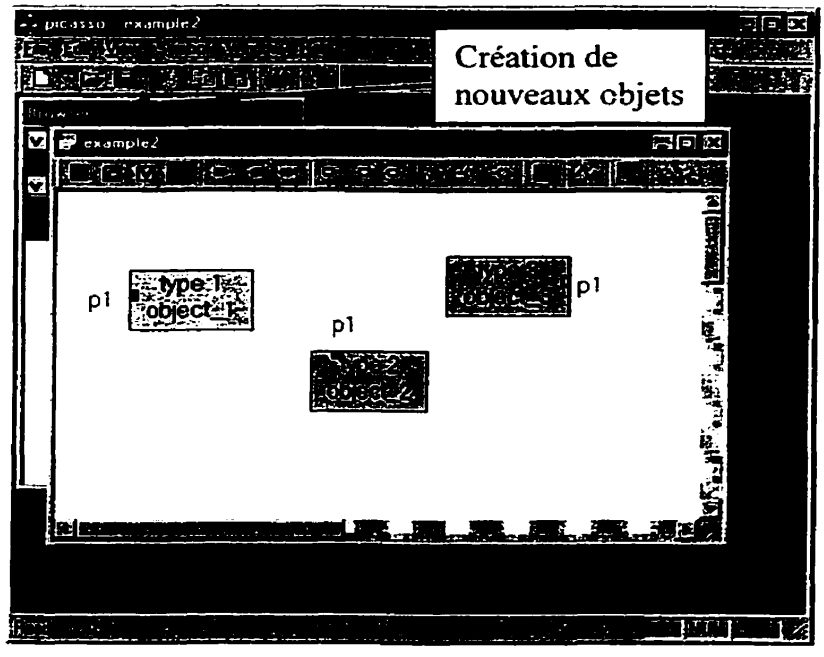

Figure 3.3 Création de nouveaux objets (approche descendante)

La déclaration de l'instance, de son interface et puis de son implantation, confirme qu'on se trouve dans une conception de type descendante. Lors d'un raffinement, un canevas de base est automatiquement créé pour tenir compte du type et des ports déclarés dans le langage utilisé (C ou VHDL). Un autre avantage de Picasso est donc de guider l'usager dans sa conception de système. À la Figure 3.4, on illustre la construction automatique du canevas de base pour l'instance « inc ». Cette instance fut définie graphiquement dans la fenêtre « *example2* »

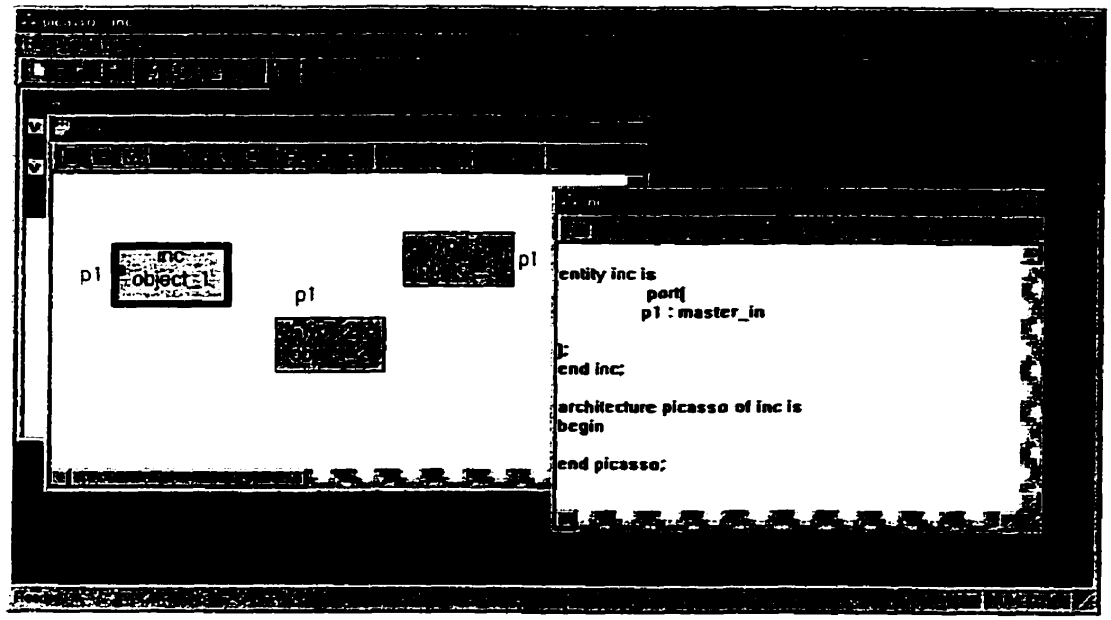

Figure 3.4 Raffinement d'un bloc

Picasso vient aider à compléter le travail de l'usager. L'usager peut compléter et raffiner son système par la suite. De plus, l'interface usager possède les fonctionnalités de base que l'on retrouve dans la plupart des éditeurs. On peut tracer des liens à plusieurs segments, on peut sélectionner plusieurs objets, on peut effacer des objets et en créer des nouveaux. On peut changer le nom des objets, déplacer les objets dans l'écran, utiliser des barres de défilement. Bref, on retrouve la plupart des avantages que la majorité des éditeurs de ce genre possèdent. Naturellement, le travail de développement pourrait se poursuivre pour ajouter encore plus de fonctionnalités à l'outil, mais là n'était pas le but premier de cette recherche.

Picasso offre un mécanisme de rafraîchissement permettant de régler les inconsistances découlant de modifications. Le changement de l'interface d'un module après son instanciation peut impliquer des conflits d'interface. Picasso garde toujours en mémoire l'interface la plus récente des modules. Il est donc possible de comparer chaque instance déjà existante avec l'interface en mémoire et de procéder aux corrections s'il y a lieu. Soit une instance branchée avec d'autres. Si on modifie la spécification du module

pour y ajouter un port, on aimerait bien que l'instance déjà existante soit mise à jour. Pour ce faire, Picasso procédera à une comparaison. Les ports inchangés ne seront pas touchés, donc ils conserveront leurs branchements, et les nouveaux ports apparaîtront. Si, lors d'une modification de l'interface, des ports sont détruits, ceux-ci disparaîtront du symbole, emportant avec eux les branchements.

Picasso est donc en mesure de supporter une approche de conception descendante. De plus, on a vu que Picasso vient aider l'usager lors de la construction de nouveaux objets par la mise au point d'un squelette d'implantation en fonction de l'interface usager. Également, Picasso dispose de mécanismes permettant de corriger rapidement les inconsistances du système provenant de modifications suivant l'approche ascendante ou descendante. Picasso possède donc toutes les qualités qu'on retrouve dans un bon environnement de saisie de spécifications.

Enfin, on constate que peu importe le langage, l'interface demeure la même. Que l'on soit en C ou en VHDL, chaque module dispose de ports qui peuvent être branchés à d'autres modules. On pourrait donc étendre ce concept à d'autres langages comme VHDL+, Verilog, Java,... et ainsi intégrer tous ces langages à Picasso. La construction de l'interface utilise toujours la sémantique du langage, en tentant d'introduire le moins d'extensions possibles. D'ailleurs, nous verrons quelles sont ces extensions dans la prochaine section.

3.2 Concepts de communication

Comme on l'a vu à la section 3.1, il existe dans Picasso trois types de ports, soit : les maîtres (*masters*), les esclaves (*slaves*) et les ports matériels. De plus, chaque port possède un sens : entrée (*in*), sortie (*out*) ou entrée et sortie (*inout*). Un port peut être considéré comme un alias sur une variable. En connectant deux ports ensembles, c'est comme si on avait deux alias référant à la même variable. Cela rend donc la

communication possible. Les ports sont typés comme les variables le sont dans la plupart des langages de programmation. Nous verrons à la prochaine section quel est le langage utilisé pour décrire les types des ports. Nous dirons pour l'instant qu'un port est typé et que lorsqu'on lit ou on écrit dans un port, on n'obtient pas nécessairement le même type. En effet, cela dépend du sens de la transaction. Comme dans le cas d'un appel de fonction, l'argument diffère souvent de la valeur de retour.

Les ports permettent d'établir différents types de communication. Les deux types de modèle sont appelés: bloquant et non bloquant. Si on se contente de lire et d'écrire sur un port, sans aucun souci de synchronisme, on parle alors de communication asynchrone, donc de type non bloquant. Considérons à titre d'exemple un système producteur-consommateur, qui procède à un sous-échantillonnage en échangeant des données au travers une variable, sans aucun mécanisme de contrôle et de verrouillage. Un tel système permet de modéliser un moteur écrivant constamment sa vitesse dans une variable. Un actionneur pourrait lire à un rythme plus lent la valeur écrite par le moteur et l'afficher. Un système transmettant la voix ou la vidéo pourrait aussi être modélisé de cette façon. Le danger avec ce type de système est la possibilité de perdre des données à cause de l'asynchronisme.

Dans le cas d'une communication un peu plus élaborée, on peut utiliser le concept de messages. Cela permet d'établir une communication bloquante. Par exemple, une fois l'écriture d'une donnée sur un port complétée, on peut envoyer un message à un destinataire qui jusque-là, était en attente. Lorsqu'il reçoit le message, il entreprend une série d'actions, comme lire la donnée envoyée et procéder à un certain traitement. Lorsqu'il a terminé, il peut renvoyer un message et ainsi l'appelant qui était bloqué peut reprendre son exécution et lire les nouveaux résultats s'il y a lieu. Cette approche ouvre la voix à des mécanismes de communication beaucoup plus complexes. On peut ainsi modéliser une communication client-serveur, de type RPC. C'est ce mécanisme qui existe actuellement dans Picasso. En offrant deux types de mécanismes, notre

environnement est plus flexible que ce qui existe actuellement commercialement (comme par exemple, Coware)

On pourrait facilement imaginer des mécanismes de communication plus complexes, par exemple définir plusieurs messages, être sensible à plusieurs messages, lancer des appels bloquants ou non. Toutes ces solutions sont actuellement étudiées et feront sans doute partie de versions ultérieures. L'introduction d'autant de mécanismes de communication nous amène à discuter du langage proprement dit.

3.3 Les modifications aux langages

Nous verrons dans cette section quels sont les éléments syntaxiques qui sont présents dans Picasso. Tout d'abord, il y a le langage pour décrire les différents types abstraits et ensuite, les extensions et fonctions apportées aux langages C et VHDL.

3.3.1 Langage de déclaration de types

Comme mentionné précédemment, les ports sont typés. Puisqu'un port se retrouve dans l'interface de communication d'un module, il est donc indépendant de l'implantation de celui-ci. On a donc besoin d'un langage pour décrire les types de ces ports qui soient indépendants de toute implantation. Pour cela, nous avons élaboré un langage qui s'inspire de XDR [15]. XDR est un langage décrivant un format permettant l'échange de données entre différents types d'ordinateurs. Il définit d'une part les représentations à suivre et d'autre part, comment les données échangées sont transformées suivant ces représentations. On se sert de ce langage dans différents contextes, comme dans les RPC (appel de procédure à distance, de l'anglais, *Remote Procedure Call*) par exemple. Dans Picasso, l'échange de données ne s'effectue pas entre différentes plate-formes, par exemple IBM et Sun, mais plutôt entre du logiciel et du matériel.

Un fait intéressant est que ce langage, développé dans ce projet de manière indépendante, a beaucoup de similitudes avec le langage en cours de développement par le groupe VSIA [24]. Ce groupe préconise une approche semblable à la nôtre, en ce sens qu'il propose un format de donnée (tout comme XDR) indépendant de l'implémentation. Par exemple, dans Picasso, un entier de 32 bits s'appelle « PIC_int », alors que dans VSIA on parle plutôt de « VSI_int8, VSI_int16, ... ». Il s'agit donc d'une similitude renforçant la viabilité de notre approche. De plus, l'adoption de la norme VSIA serait relativement aisée.

Le langage Picasso peut se décomposer en deux parties. Tout d'abord, il décrit les types qui sont prédéfinis dans le système. Pour l'instant, il n'y a que trois types, mais il serait très simple d'en intégrer d'autres. Un "PIC_int" correspond à un entier de 32 bits signé. Un "PIC_bool" représente un seul bit (vrai ou faux). Un "PIC_char" représente un entier de 8 bits signé. À l'intérieur de Picasso, chaque type correspond à une classe C++ qui dérive d'une classe abstraite parente. Il suffirait donc de créer d'autres descendants de la classe parente pour automatiquement créer d'autres types. Une classe de type doit être en mesure de remplir certaines obligations, comme avoir une représentation du type pour chaque langage utilisé.

La seconde partie du langage permet la création de types complexes. Il s'agit des types « PIC_struct » et « PIC_typedef ». « PIC_typedef » permet la création d'alias et aussi de tableaux. Il permet de définir des tableaux à N éléments dont le type a déjà été défini. « PIC_struct » permet la création d'enregistrements contenant n'importe quel type de membre. On peut utiliser les membres de base ou d'autres membres complexes construits par l'usager. Par exemple, une structure de deux entiers pourrait être décrite comme à la Figure 3.5.

```
PIC_Struct sum{
        PIC_int x,y;
}
```

Figure 3.5 Exemple de déclaration d'une structure

3.3.2 Extension au langage C

On peut maintenant aborder le langage utilisé pour la description du logiciel. Il s'agit du langage C augmenté de certaines extensions permettant ainsi la communication et la déclaration de ports. La première extension au niveau du langage C se situe au niveau de la notion de zone, (*scope*). Dans le langage C, on a toujours une section appelée « *main* » qui constitue le point d'entrée du programme. Puisque les spécifications d'un système embarqué s'expriment souvent en termes de processus élémentaires. cette notion de zone peut être limitative. Nous avons donc introduit le concept de processus élémentaire au niveau du langage C. Par conséquent, le « *main* » a été retiré et remplacé par le mot clé « *thread* ». Cela permet d'avoir plusieurs contextes d'exécution au lieu d'un seul. Le code de la Figure 3.6 contient deux processus élémentaires. Le premier est utilisé comme client et demande à un autre module d'additionner deux nombres. Le second effectue une tâche quelconque en parallèle.

```
port {
        my_port : master_inout sum => PIC_int;
}


thread client{
   PIC_int result;

   Write(my_port.x,40);
   Write(my_port.y,50);
   Start(my_port);
   result=Read(my_port);

   Pause;
}

thread second_thread{
        /* do something*/
}

init {
        /* init code*/
}
```

Figure 3.6 Client de l'additionneur en logiciel

Une autre zone ajoutée est celle appelée « port ». Elle sert à déclarer l'interface d'un module. Chaque port présent dans cette section se comporte comme une variable. On peut donc le manipuler à l'aide des instructions fournies. Un port est toujours composé d'un nom, suivi des deux points et de sa classe. Il s'agit soit de maître (*master*) ou esclave (*slave*) accompagné du sens du transfert de données: entrée et/ou sortie(*in, out, inout*). Enfin on précise le type des données envoyées et/ou reçues. Ces types de données sont décrits avec le langage défini préalablement. Dans l'exemple de la Figure 3.6, il n'y a qu'un port appelé « my_port ». C'est un port maître d'entrée/sortie. Le type des données envoyées à l'instruction *start*, est « sum », soit le type défini précédemment (Figure 3.5) comme étant une structure de deux entiers. Il s'agit donc des deux valeurs à additionner. Le type de la donnée retournée dans *result*, lors d'une lecture, est un « PIC_int ». Il s'agit de l'entier qui contiendra la somme. Le port maître permet donc d'envoyer le message *start*, et le port slave

démarre son exécution à la réception du message. C'est ce qui différencie les deux types de port.

Une fois les ports déclarés, on retrouve (toujours à la Figure 3.6) une ou plusieurs zones appelés « *thread* ». Chacune de ces zones définit un contexte d'exécution particulier. On ne peut pas prévoir dans quel ordre chaque processus élémentaire sera appelé, mais la séquence reste uniforme. C'est donc dire que pour deux processus élémentaires de même priorité, on ne peut pas ré-exécuter l'un d'eux sans exécuter l'autre une fois, sauf s'il est bloqué. Dans chaque processus élémentaire, on dispose de la fonction « pause ». Chaque fois que cette fonction est appelée, le contexte d'exécution actuel est conservé, puis l'ordonnanceur est appelé pour passer à un autre processus élémentaire. Ce mécanisme est comparable à celui présent en VHDL, lors de l'exécution d'un *« wait statement »*. C'est donc dire que la seule différence avec une bibliothèque de processus élémentaires conventionnelle est l'absence d'une minuterie pour amorcer le changement de contexte de façon « aléatoire ». Les processus élémentaires sont exécutés de façon uniforme. C'est-à-dire que lorsque le processus élémentaire #1 exécute la fonction pause, il passe la main au processus élémentaire numéro #2 et ainsi de suite. À la fin, le dernier processus élémentaire va repasser la main au processus élémentaire #1 qui va reprendre son exécution à l'endroit où il l'a laissé.

Les processus élémentaires à l'intérieur d'un même module peuvent communiquer par variables globales. Puisque chaque processus élémentaire passe la main à un autre et qu'il n'y a pas de préemption, on peut supposer raisonnablement qu'il n'y aura pas de problèmes de synchronisation. On voit donc qu'il n'est pas si important d'implanter des sémaphores. Supposons, par exemple, une liste implantée par une variable globale. Selon sa fonctionnalité, un processus élémentaire pourrait effectuer tous les changements, puis ensuite appeler la fonction pause. À l'opposé, une

mauvaise stratégie de la part du concepteur serait de commencer les modifications et d'appeler la fonction pause.

Un processus élémentaire spécial nommé « *init* » est également présent. Ce processus élémentaire est appelé une seule fois au tout début, avant même d'appeler l'ordonnanceur. Il permet, par exemple, de fixer des valeurs par défaut sur les ports et d'initialiser les diverses variables globales du système.

Cette façon de diviser le système en module et en processus élémentaires implique qu'on a deux niveaux de granularité différents dans Picasso. Les processus élémentaires qui communiquent entre eux très fortement et qui ont une très haute cohésion devraient être préférablement dans le même module logiciel. Les processus élémentaires qui communiquent moins entre eux devraient être placés dans des modules séparés. Cela permet d'explorer différentes architectures en utilisant un ou plusieurs microprocesseurs pour chaque instance. Si plusieurs instances sont réunies sur le même microprocesseur, les différents processus élémentaires qui les composent seront alors rassemblés. Cette granularité variable est donc un autre avantage de Picasso et elle facilite l'exploration de système.

Pour l'instant, les analyseurs grammaticaux de Picasso sont encore à l'état expérimental. Aucune analyse selon le contexte n'est effectuée. Il peut donc surgir un problème de nomenclature avec les variables globales. En effet, supposons qu'on crée deux instances du même type et que l'on décide d'associer les deux instances au même microprocesseur. Lors de l'étape de rassemblement du code, on se retrouve avec la même variable globale deux fois. La solution retenue pour l'instant est de donner des noms différents à ces variables et à changer le nom partout dans les deux modules avant de générer le module final. Toutefois, pour implémenter cette solution, il faut tout d'abord déterminer qu'on a une variable globale. Cela ne peut se faire que par une évaluation du contexte de la variable, à savoir si elle se trouve dans une

fonction, un processus élémentaire, une structure,.... ou au niveau global. Puisqu'il s'agit d'une tâche complexe, la solution a plutôt été d'adopter un préfixe unique pour designer les variables globales. Par exemple, pour créer une variable globale de type entier nommé « alpha », on doit déclarer « int GLOBAL_alpha ». Naturellement, toutes les expressions suivantes vont utiliser « GLOBAL_alpha ». Cela permet facilement à Picasso d'identifier les variables globales et de les traiter convenablement.

Si on regarde maintenant au niveau du contenu d'un processus élémentaire, on dispose de toutes les fonctionnalités déjà présentes dans le C, en plus de certaines extensions permettant de travailler avec les ports de communication. On dispose des fonctions *Read*, *Write* et *Echo*. *Read* retourne la valeur d'un port. À la Figure 3.6, la commande *Read* permet de lire le résultat de la somme et le placer dans la variable « result ». La commande *Write* envoie une valeur sur un port. Dans l'exemple, on l'utilise deux fois pour envoyer les deux opérandes à additionner. La commande *Echo* quant à elle permet d'aller relire ce que l'on a déjà écrit. En effet, il faut garder en tête que pour un port de type entrée et sortie, la donnée écrite n'est pas du même type que celle qui est lue, car deux canaux distincts sont créés. L'avantage de la commande *Echo* est qu'elle permet de ne pas garder une copie locale de la donnée. Dans le cas de données de grandes tailles, cela évite d'avoir une copie d'information entre la mémoire du processeur et le port.

Il est important de noter que l'on peut fournir un sous-ensemble du type du port (pour les tableaux et les structures) aux trois fonctions précédentes. Par exemple, nous aurions pu utiliser la syntaxe à la Figure 3.7.

```
sum beta; /*declare la variable*/
/*initialiser beta*/
Write(my_port,beta);
```

Figure 3.7 Exemple d'utilisation d'une structure

Si « beta » avait été un tableau, on aurait pu utiliser l'indice dans la fonction pour l'adresser. La notion d'accès à un sous-ensemble d'un tableau en précisant un intervalle n'est pas permise, n'étant supportée qu'en VHDL uniquement. On ne peut accéder à un tableau que élément par élément ou en accédant le tableau dans son ensemble.

Pour envoyer un message et ainsi amorcer une communication, on utilise la commande « *Start* ». Cette commande envoie un message le long de la connexion et assume qu'une fonction prendra en charge le message. Elle est toujours envoyée par un port maître et reçu par un port esclave. Ceci explique la raison pour laquelle cette nomenclature est utilisée pour décrire les ports. Lorsque la fonction du coté esclave (*slave*) termine, un message est envoyé automatiquement par le système pour indiquer que le maître ayant effectué le *Start* peut poursuivre son exécution.

La fonction du côté esclave est identifiée par le mot clé *slave*, au lieu du mot *thread* habituel. En fait, il s'agit bel et bien d'un processus élémentaire, à l'exception qu'il est désactivé lors de l'initialisation du système. Donc, tant que le message *Start* n'est pas reçu, l'ordonnanceur l'ignore. Quand le message est reçu, ce processus élémentaire redevient actif et donc peut être ordonnancé. Lorsqu'il terminera, il redeviendra désactivé, jusqu'à la prochaine requête. On peut illustrer ceci à l'aide d'un autre exemple. Pour compléter notre client qui veut additionner deux valeurs, nous allons créer un autre module. Ce module fera l'addition. Le code est illustré à la Figure 3.8. On crée un port « *receptor* » qui est un esclave, donc qui reçoit le message *start*. Les types de données sont les mêmes que pour le client pour être compatibles. On déclare ensuite le processus élémentaire esclave qui sera associé à ce port. Lorsqu'il est exécuté, lors de l'appel du client, il lit les deux valeurs envoyées par le client et envoie la somme. Comme il termine, il envoie un signal pour débloquer le client et l'ordonnanceur le place en attente jusqu'au prochain appel.

```
port {
        receptor:slave_inout PIC_sum => PIC_int;
}

slave receptor {
    Write(receptor,Read(receptor.x)+Read(receptor.y));
}
```

Figure 3.8 Additionneur en logiciel

À l'aide du même principe, on peut désactiver un processus élémentaire qui entreprend une opération d'entrée/sortie. Le cas le plus évident est le « *Start* ». Tant que l'esclave n'a pas terminé, le processus élémentaire maître est en attente. Pour être efficace, on peut donc le désactiver. Lorsque le système recevra le message indiquant la fin de l'esclave, alors l'ordonnanceur remettra le processus élémentaire ayant fait l'appel *Start* comme étant actif. On pourrait appliquer le même principe aux opérations d'entrée/sortie, quoiqu'ici elles soient très rapides (comparable à un accès mémoire). Ceci devra probablement être considéré lorsque d'autres mécanismes de communication seront implémentés.

On note donc qu'il y a une forte cohésion entre la bibliothèque de processus élémentaires et la bibliothèque de communication. Ces deux entités doivent toujours s'échanger de l'information. Puisque le module de communication fut conçu de toutes pièces, il va de soi qu'il était plus simple de créer notre propre bibliothèque de processus élémentaires, plutôt que d'en utiliser une existante. Un autre argument en faveur de notre approche est que la plupart des bibliothèques portables de processus élémentaires déjà existantes utilisent un système d'exploitation. Une solution aurait été de porter un système d'exploitation pour l'ensemble des différents processeurs utilisés, ce qui semblait un projet d'envergure. Nous avons donc opté pour une solution plus simple. On peut donc voir le tout comme un petit système d'exploitation maison. Une fois que le prototype de Picasso aura atteint un certain niveau de maturité, nous pourrons sûrement utiliser des composants plus standards, offrant donc

du même coup un gain en fonctionnalité, et un niveau de standardisation facilitant l'apprentissage.

3.3.3 Extension au langage VHDL

Nous allons maintenant examiner le cas du VHDL. Contrairement au C, le VHDL contient déjà différentes zones dans une description. Les plus connues, et celles que nous allons aborder, sont « *entity* » et « *architecture* ». L'entité est la partie qui déclare l'interface d'un module VHDL. Elle contient la définition des ports d'entrés et de sorties. C'est donc dans l'entité que nous procédons à certaines extensions pour rendre l'entité compatible avec Picasso. Picasso a comme avantage de supporter des descriptions mixtes pour les systèmes matériels. C'est-à-dire qu'il peut contenir des ports VHDL, maîtres et esclaves dans la même déclaration. La déclaration des ports se fait de la même façon que celle présentée pour les extensions du langage C. L'encadré suivant donne un exemple d'un module qui additionne deux valeurs, mais cette fois en VHDL. La syntaxe pour déclarer les ports maîtres ou esclaves est la même qu'en C, et elle s'insère aisément dans le VHDL standard. Dans l'exemple de la Figure 3.9, on note que dans la même entité, on déclare un port esclave et deux ports standards en VHDL.

```
entity adder_vhdl is
        port(
                value : slave_inout PIC_sum => PIC_int;
                clk:in std_logic;
                reset:in std_logic
        );
end adder_vhdl;

architecture picasso of adder_vhdl is
begin

value:slave
variable i:PIC_int;
variable j:PIC_int;
variable k:PIC_int;
begin
        Comm_Attrib(clk,reset);
        i:=Read(value.x);
        j:=Read(value.y);
        Write(value,i+j);
end slave;

end picasso;
```

Figure 3.9 Additionneur en matériel

Examinons maintenant l'architecture. Habituellement, elle se compose de plusieurs processus. On peut considérer l'ensemble des processus fonctionnant en concurrence. On peut donc affirmer que ce comportement est semblable aux processus élémentaires et donc qu'on n'a pas besoin de modifier le langage pour les supporter. On a toutefois ajouté le mot clé *slave* qui peut remplacer le mot *process*, pour permettre la création d'esclaves comme pour le C. Naturellement, nous n'utilisons pas de listes de sensibilité pour ce type de processus, celle-ci étant remplacée par le nom du port esclave.

De même, on ne peut pas utiliser de listes de sensibilité pour les processus utilisant les extensions de Picasso, car ils sont considérés comme des processus séquentiels. À la place, ces processus doivent toujours débuter par une ligne déclarative. Il s'agit de l'instruction « comm_attrib ». Cette instruction est nécessaire pour la phase de génération finale du système. En effet, les instructions de

communication ajoutées en extension au VHDL se traduisent souvent par un ensemble d'instructions VHDL standard. On doit alors synchroniser ces différentes instructions. Par exemple, les boucles d'attente actives doivent être validées à chaque coup d'horloge. Physiquement, cela se traduira avec des registres conservant l'état du système. L'énoncé « Comm_attrib » vient donc préciser quel signal d'horloge et quel signal de remise à zéro sera utilisé par les processus. On se limite à une horloge par processus, car la plupart des outils de synthèse imposent cette contrainte[1]. Par contre, on peut utiliser des horloges différentes pour différents processus. En utilisant les ports de Picasso, on peut ainsi transférer des données dans un système roulant avec différentes horloges et être assuré que la cohérence de la communication soit respectée.

Actuellement, on suppose que la réinitialisation du module se produit lorsque le port définit comme remise à zéro passe au niveau '1'. Il serait facile d'ajouter un troisième paramètre à l'énoncé « Comm_attrib » permettant de spécifier explicitement le niveau (zéro ou un) qu'on désire utiliser pour effectuer la remise à zéro.

Finalement, tout comme dans la partie logicielle, les commandes *Echo*, *Read* & *Write* sont aussi ajoutées au VHDL. Leur définition et leur fonctionnement sont les mêmes que pour le C.

Notons en terminant qu'on aurait pu concevoir un langage textuel décrivant l'interface graphique de connexions. En effet, un objet Picasso contient des instances d'objets et des connexions. On aurait donc pu créer un langage textuel qui offrirait ces possibilités. Cela permettrait à l'usager de créer son système en employant l'une ou l'autre des formes. Cela permettrait aussi d'ajouter plus facilement certaines

---

[1] Synopsys offre des opérations multi-clocks dans BC. Cela semble par contre encore à l'état expérimental.

possibilités à l'outil, comme la création de tableaux de ports. En effet, on pourrait vouloir créer un port qui est en fait un ensemble de ports. Cela augmenterait la lisibilité du code, car au lieu de manipuler chaque port par un nom distinct, on pourrait l'adresser par son index. L'exemple du système de télécommunication du chapitre 6 fera ressortir ce point.

3.4 Sélection des processeurs

La dernière étape consiste à créer les différents processeurs qui composeront la partie logicielle du système et d'associer chaque instance logicielle à un de ceux-ci. Chacun de ces processeurs est en fait un sous-système qui est modifiable par Picasso. Il contient un microprocesseur, des mémoires et une interface commune qui permet à Picasso d'y brancher les circuits qu'il génère. Nous appelons ces sous-systèmes des Ichip. Dans l'outil Picasso, on peut créer des Ichip et associer la ou les instances logicielles à un Ichip. Des instances logicielles peuvent être reliées au même Ichip ou à des Ichips différentes. S'il s'agit de la même instance, le code sera aggloméré ensemble et la communication sera implantée en logiciel. Dans le cas d'instances différentes, la communication se fera par bus entre les instances de processeurs. Cela conduit à un avantage important, soit l'exploration d'architectures. Picasso permet d'ajouter ou d'enlever facilement des microprocesseurs dans le système et d'associer facilement des morceaux de logiciel à tel ou tel microprocesseur. À partir d'une configuration donnée, il peut générer tout le système, prêt pour la simulation et la synthèse.

Dans ce chapitre, nous avons vu comment l'outil Picasso permet à l'usager de spécifier son système. De nombreux avantages d'utiliser l'outil ont été présentés, mais il reste également encore de nombreux points à améliorer. Nous verrons au chapitre suivant comment l'outil, à partir de l'ensemble des spécifications, parvient à les transformer pour générer une implémentation.

Chapitre 4 – Architecture cible

Dans ce chapitre, nous verrons le fonctionnement de l'architecture ciblée. Puisqu'on ne dispose que d'un seul mécanisme de communication, le principe de fonctionnement de l'architecture générée est toujours le même. L'architecture est toujours adaptée en fonction des spécifications. La série de transitions qui permettent de passer de la spécification de haut niveau jusqu'à l'architecture sera vue au chapitre suivant.

Pour mieux expliquer l'architecture, supposons un ensemble de processus élémentaires logiciels (maîtres et esclaves) exécutés sur un processeur. Ces processus élémentaires communiquent avec un ensemble de modules matériels (également maîtres et esclaves). Le cas où il y a plusieurs processeurs sera vu plus loin. La Figure 4.1 montre l'architecture matérielle du système final. Il y a trois grandes parties, soient le « *Ichip* » (pour *interface for chip)*, le « *archi* » (pour *communication architecture*) et les différents modules matériels résultant de la synthèse du VHDL spécifié par l'usager. Nous allons examiner tour à tour le fonctionnement de ces différentes parties.

4.1 Présentation générale de l'architecture du système

Le *Ichip* représente une instance de processeur. Il contient un processeur, sa mémoire ROM et RAM, une interface de communication par mémoire partagée et toute la logique d'interconnexion et de contrôle. L'interface se comporte comme une mémoire par rapport au microprocesseur. Pour le microprocesseur, l'interface est une mémoire associée à une plage d'adresses réservées. Toute lecture ou écriture du processeur dans cette plage est envoyée directement à l'interface. Le module *archi* doit donc offrir une interface semblable à une mémoire.

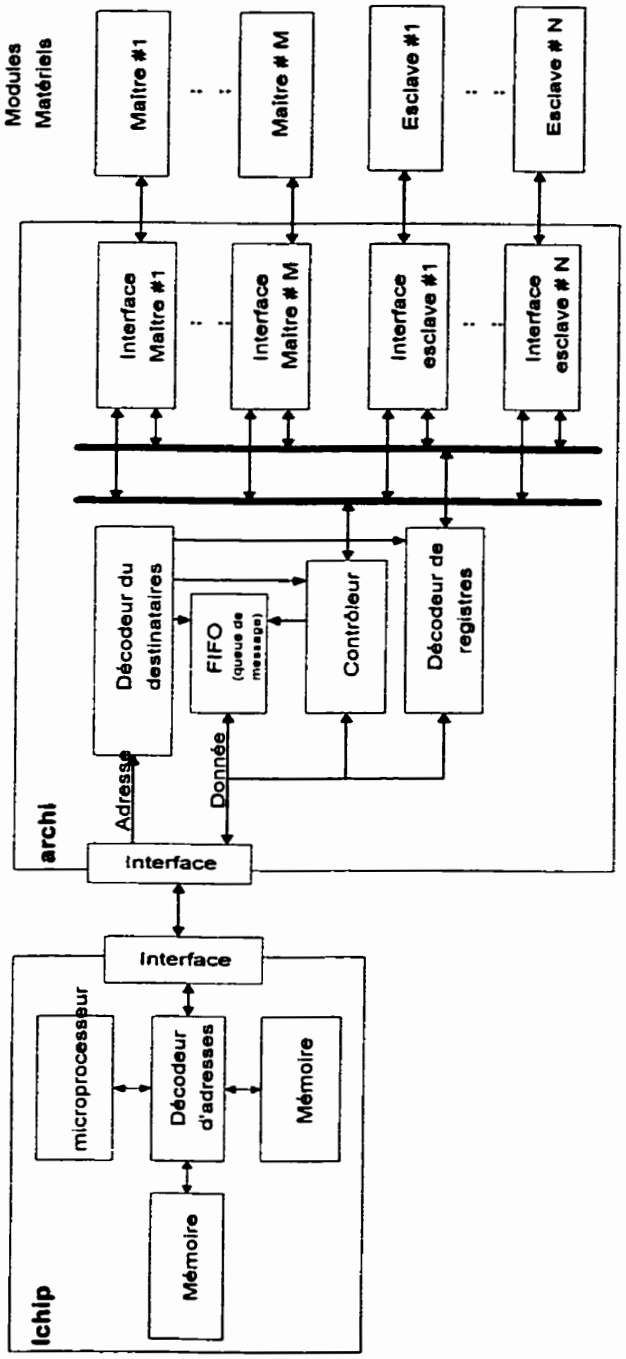

Figure 4.1 Architecture du système

D'un point de vue logiciel, le *Ichip* est un pilote qui connaît les instructions binaires (assembleurs) pour accéder à l'interface. Ces instructions servent à implanter les diverses extensions du C. En résumé, le *Ichip* est une entité capable de transférer des données entre le logiciel et le matériel et vice versa. Pour le *Ichip,* les données n'ont aucune interprétation particulière. C'est-à-dire qu'il ne s'agit que d'adresses mémoires accompagnées de données. La « signification » de ces adresses et données viendra du module « *archi* ».

Le module *archi* apporte une signification aux données. Il agit comme un bus dédié entre le *Ichip* et les différents blocs matériels.

Quant aux blocs matériels attachés au *archi*, on doit implanter leurs ports par des registres. On traduit donc les types de haut niveau en type VHDL. Chaque module convertit donc ses ports en un ensemble de registres. Pour implanter les transactions maîtres/esclaves via la commande *start*, (section 3.3), on ajoute des signaux de contrôle à chaque port pour indiquer l'état et commander les processus associés. Enfin, on identifie chaque port par un entier unique. Cela permet au logiciel comme au matériel de s'y référer.

Dans le module *archi*, le premier bloc à examiner est le décodeur de destinataire. Celui-ci détermine deux adresses particulières à l'intérieur de la plage d'adresses fournie par le *Ichip*. Une adresse sera destinée au contrôleur, l'autre au FIFO. Toutes les autres adresses seront considérées comme appartenant à un registre d'un bloc matériel et seront traitées par le décodeur de registre. Ainsi, lorsqu'une adresse sera détectée sur l'interface, le décodeur d'adresse activera le bon bloc (FIFO, contrôleur ou décodeur de registre).

Lorsque le décodeur de registre est activé, un deuxième décodage d'adresse entre en jeu. Il s'agit de déterminer à quels ports de quel module matériel le microprocesseur tente d'accéder. Puisque chaque port est représenté par un registre, on peut accéder aux

données du bon registre. Cela est possible car chaque port de chaque module matériel possède une adresse unique.

Chaque port du coté matériel est attaché au *archi* par une interface. Cette interface permet le transfert des données et des signaux de contrôle. Il y deux types d'interfaces : une pour les ports maîtres et l'autre pour les ports esclaves. Ces interfaces permettent de rattacher correctement le module au reste du *archi*.

Lorsque l'adresse correspond plutôt à celle du contrôleur dans le *archi*, celui-ci est activé. La donnée correspond au numéro d'identification du port auquel on veut envoyer un message de contrôle. Il faut se rappeler que chaque port possède un identificateur unique. Le contrôleur peut donc avertir la bonne interface, en lui disant de commencer un traitement matériel, ou indiquer qu'un traitement logiciel est terminé.

La dernière boîte qui reste à examiner est le FIFO. Celui-ci est utilisé pour maintenir la liste des modules matériels qui veulent communiquer avec le logiciel. Pour l'instant, notre architecture ne supporte pas d'interruption. Le logiciel doit donc scruter (en anglais, *polling)* pour détecter les messages provenant du matériel. Lorsqu'un module matériel maître veut démarrer un processus élémentaire esclave, ou quand un module matériel esclave a terminé, le numéro d'identification du bloc matériel est placé dans le FIFO. À intervalle régulier, l'OS (pour système d'exploitation, de l'anglais, *Operating System*) lit les valeurs du FIFO. Il peut alors ajuster correctement les états des différents processus élémentaires associés aux identificateurs lus. L'avantage du FIFO est qu'on n'a pas à scruter chaque port. À chaque fois que l'OS passe à cette phase, il suffit de regarder si un identificateur se trouve dans le FIFO. S'il n'y en a pas, alors on sait qu'aucun périphérique matériel n'a d'information à communiquer, ce qui arrive fréquemment. Puisque le FIFO se trouve à une adresse mémoire, on sait en un seul accès mémoire quels modules sont prêts à communiquer.

Lorsqu'un processus élémentaire active un module matériel esclave, il se produit les événements suivants. Tout d'abord, l'OS identifie le numéro du port esclave correspondant à celui qu'il désire éveiller. Puis il envoie ce numéro au contrôleur du *archi*, en écrivant à son adresse. Le contrôleur peut ensuite envoyer les bons signaux de contrôle au bon port du bon module. Cela démarre le module matériel esclave. Lorsque ce dernier a terminé, il avertit le contrôleur du *archi*. Celui-ci place son numéro d'identification dans le FIFO. Lorsque l'OS ira lire le FIFO et verra l'identificateur qu'il avait envoyé auparavant, il pourra débloquer le processus élémentaire associé.

Dans la situation contraire, si un module matériel maître réveille un processus élémentaire esclave, on se retrouve avec un scénario très similaire. Le module maître commence par placer son numéro d'identification dans le FIFO. Au moment où l'OS lira le FIFO, il trouvera le numéro d'identification. L'ordonnanceur de l'OS possède une table qui associe chaque identificateur à son processus élémentaire. Il pourra donc réveiller le processus élémentaire esclave associé. Ce processus élémentaire sortira de son état « endormi » et pourra être sélectionné par l'ordonnanceur. Lorsque le processus élémentaire aura complété son exécution, l'ordonnanceur, sachant qu'il s'agit d'un processus élémentaire esclave, renverra son numéro d'identification au contrôleur du *archi*. Celui-ci pourra renvoyer les bons signaux au module matériel pour le débloquer.

Cela complète l'architecture lorsqu'il n'y a qu'un seul processeur. Dans le cas contraire, le système demeure très semblable. En effet, les interfaces des modules matériels présentes dans le *archi* sont conçues pour être connectées dos-à-dos. La Figure 4.2 montre un système où il y aurait deux *Ichip* (A et B) et un module matériel spécifique attaché au module A. Dans ce cas-ci, puisqu'il n'y a qu'un seul module branché sur chaque archi, leur réalisation serait très simple. En effet, il n'y aura par exemple qu'une seule interface de communication. Le FIFO sera ramené à un seul registre (donc de taille unitaire) et ainsi de suite pour les autres parties.

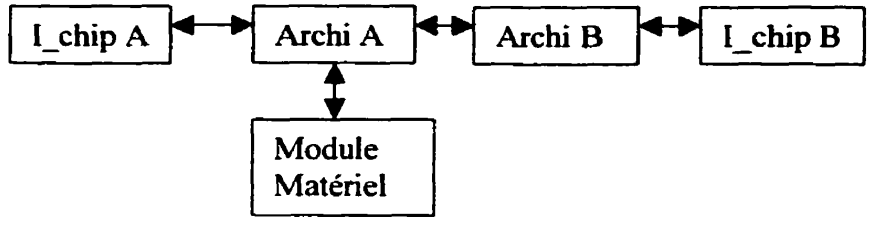

Figure 4.2 Architecture à deux Ichips

Voyons plus en détails, au niveau implantation, le fonctionnement de chacun de ces blocs.

4.2 Le matériel du bloc *Ichip*

Comme on l'a mentionné, le bloc *Ichip* est comme un système virtuel incluant un microprocesseur, les mémoires RAM et ROM et un port permettant d'y brancher un bloc *archi*. Dans notre cas, le microprocesseur sélectionné fut un i960 de Intel [1]. Pour celui-ci, la description du *Ichip* que nous utilisons a été dérivée d'un exemple qui était livré avec Seamless. Nous avons modifié cet exemple pour qu'il remplisse nos besoins.

Le fichier « i_chip.vhd » (annexe p.130) contient la description du système. L'entité décrit l'interface qui doit être branchée au bloc *archi*. Pour le moment, c'est le seul moyen de communication avec l'extérieur du *Ichip*. Le

Tableau 4-1 présente le nom de chaque signal, son type et une courte description de sa fonctionnalité.

Tableau 4-1 Ports du Ichip

Nom	Type	Description
W_R_bar	inout std_logic	Indique si on écrit ou on lit des données sur le bus
archi_addr	out std_logic_vector(27 downto 0);	Bus d'adresse
LAD	inout std_logic_vector(31 downto 0);	Bus de données
archi_cs_bar	inout std_logic;	Indique si le port est activé ou non (haute impédance)
OE_bar	inout std_logic;	Indique si la sortie mémoire est activée
WE_bar	inout std_logic;	Indique si l'écriture est activée
CLK	inout std_logic;	Horloge du *Ichip*
RESET	inout std_logic	Remise à zéro du *Ichip*

Si on regarde l'architecture du *Ichip*, on s'aperçoit qu'il est constitué de blocs de base : le i960, la mémoire formée de 8 composants couvrant 2 banques de 32 bits et la logique permettant l'interconnexion des différents composants. Pour plus de détails, le lecteur est prié de consulter l'annexe Annexe 17 -.

Actuellement cette description demeure fixe, mais comme on l'a déjà mentionné, un générateur *Ichip* pour le matériel est envisageable. Ainsi, on pourrait procéder aux diverses modifications locales, comme par exemple, changer le type ou le nombre de mémoires. Également, il faudra développer l'interface de connexion pour avoir un bus d'adresse qui puisse entrer et interroger la mémoire locale. Pour cela, un système d'arbitrage de la mémoire sera nécessaire pour partager la mémoire entre le client extérieur au Ichip et le microprocesseur.

4.3 Le logiciel du bloc *Ichip*

La partie logicielle du *Ichip* est en fait un API que nous avons bâti de zéro et qui est portable de processeur en processeur. Il comprend le système de communication du Ichip et la bibliothèque de processus élémentaires. Seul un petit noyau doit être récrit en assembleur, ce qui diminue l'effort pour le porter. Au Tableau 4-2, on retrouve les fonctions présentes, au Tableau 4-3, les structures de données et enfin au Tableau 4-4, on présente les principales macros utilisées.

La bibliothèque de processus élémentaires est semi-préemptive. Comme dans tous les systèmes préemptifs, chaque processus élémentaire possède son propre contexte, sa propre pile et peut être interrompu sur n'importe quelle instruction. Par contre, il n'y a pas d'interruption pour passer d'un processus élémentaire à l'autre. C'est le processus élémentaire lui-même qui décide de se suspendre et d'appeler l'ordonnanceur, qui partira un autre processus élémentaire. Cela laisse aux concepteurs le soin de préciser dans le code où se feront les changements de contexte, ce qui a pour effet de diminuer considérablement ce nombre de changement et donc de diminuer la perte de temps causée par ces changements. L'inconvénient est de prévoir judicieusement et à l'avance, les endroits où placer les instructions pauses.

Tableau 4-2 Fonctions du Ichip

void change_thread();	Appelle l'ordonnanceur pour sauvegarder le contexte du processus élémentaire et passer à un autre
void send(int pCopro);	Envoie un identificateur au contrôleur du archi
void io_polling();	Lit les données du FIFO du archi
void Lock(int i);	Verrouille un processus élémentaire pour empêcher qu'il soit choisi par l'ordonnanceur.
void Unlock(int i);	Déverrouille un processus élémentaire
void init_thread();	Initialise les contextes d'exécution des processus élémentaires
void StartEngine();	Démarre l'ordonnanceur

Tableau 4-3 Structure de données du Ichip

typedef void (*Tthread_func)();	Type utilisé pour définir l'entête des processus élémentaires
extern int GlobalThread;	Identificateur du processus élémentaire courant
extern jmp_buf thread_env[nb_thread];	Tableau de structures *jmp_buf* permettant de conserver les registres de chaque processus élémentaire

Tableau 4-4 Macros du Ichip

#define polling_mem *(int*)0x60000000	Adresse permettant de tirer du FIFO l'information de communication propre à chaque Ichip.
#define ctrl_mem *(int*)0x50000000	Adresse à utiliser pour les messages de contrôle destinés au matériel. Propre à chaque Ichip.
#define true 1	Valeur booléenne
#define false 0	Valeur booléenne
#define Pause if (!setjmp(thread_env[GlobalThread])){change_thread();}	Macro permettant de conserver l'état d'un processus élémentaire et d'appeler l'ordonnanceur

Mentionnons également un point important en ce qui concerne le logiciel du bloc Ichip soit l'utilisation de variables globales. Celles-ci sont nécessaires car les informations sur chaque processus élémentaire ne peuvent pas être stockées dans le contexte local d'un des processus élémentaires. Il faut donc utiliser un espace mémoire qui demeure intouchable et qui n'est dans aucune pile. La zone de stockage des variables globales répond parfaitement à ce besoin.

4.4 Fonctions send, init_thread, lock et start_engine

Pour bien expliquer le fonctionnement de certaines fonctions de l'API et l'interaction des divers composants, nous allons reprendre l'exemple que nous avons expliqué au chapitre 3, à la section 3.3 . Il s'agit du client réalisé en logiciel. Dans la Figure 4.3, nous avons traduit la spécification de haut niveau en utilisant l'API. Le chapitre suivant illustrera comment cette transformation peut être faite automatiquement par Picasso.

```
port {
my_port : master_inout PIC_sum =>
PIC_int;
}


thread my_thread{
   PIC_int result;

   Write(my_port.x,40);
   Write(my_port.y,50);
   Start(my_port);
   result=Read(my_port);

   Pause;
}
```

- Basé sur la figure 3.6

```
/* generated by picasso, 1999, 2000 */
#include "data_type.h"
#include "i_chip.h"


#define object_2_my_port_data_in (*(PIC_int*)1073741824)
#define object_2_my_port_data_out (*(PIC_sum*)1073741828)
#define object_3_receptor_data_in (*(PIC_sum*)1073741836)
#define object_3_receptor_data_out (*(PIC_int*)1073741844)

void my_thread_thread()
{
 for(;;){
   PIC_int result;

   object_2_my_port_data_out.x=40;
   object_2_my_port_data_out.y=50;
   send(1);
   Pause;

   result=(object_2_my_port_data_in);
   Pause;
 }
}

void init() {}

Tthread_func thread_func[]={my_thread_thread};

Void main()
{
        init_thread();

        /*lock the slave thread*/
        Lock(1);

        StartEngine();
}
```

Figure 4.3 Transformation du client logiciel par Picasso (avant & après).

Tout d'abord, à la Figure 4.3, on remarque que les ports sont transformés en « #define » pointant à des adresses spécifiques. Il s'agit d'adresses qui permettent d'accéder aux registres matériels implantant ces ports. L'intervalle dans laquelle ces adresses ont été sélectionnées n'appartient donc pas à la mémoire, mais au module *archi*. Les commandes *Read* et *Write* sont donc transformées pour manipuler directement ces adresses.

Le fichier généré contient un tableau appelé « thread_func ». Il s'agit du tableau créé par l'étape de génération et qui contient un pointeur sur chaque processus élémentaire ou processus élémentaire esclave.

On note que le « *main* » se trouve dans le fichier associé au *Ichip*. Il est là car son contenu est variable. Cela permet d'avoir un API qui n'a pas besoin d'être régénéré à chaque fois par Picasso. La première tâche du « *main* » est d'initialiser les processus élémentaires par la commande « init_thread ». Pour chaque processus élémentaire valide du système, on doit réajuster son pointeur de contexte et initialiser son entrée dans un tableau de communication interne (ID_com). Le tableau de contextes et le tableau de pointeurs sont des variables globales. C'est ce qui sert de contexte (pile) pour chaque processus élémentaire.

Concernant le tableau de communication (ID_com) mentionné précédemment, il est utilisé pour mémoriser les processus élémentaires qui sont en attente. Un numéro d'identification est envoyé lorsqu'un module matériel esclave termine son exécution. Aussitôt qu'un processus élémentaire utilise la commande *start*, on note dans ce tableau, pour le processus élémentaire donné, quel identificateur va le relancer. Donc, quand on reçoit un message indiquant qu'un module matériel esclave a terminé, on peut déverrouiller le bon processus élémentaire selon l'identificateur reçu.

Ensuite, on constate que le *main* appelle la fonction « lock » avec des identificateurs. Ces identificateurs correspondent aux processus élémentaires esclaves en logiciel et sont similaires à ceux vus précédemment. Cela permet de verrouiller ces processus élémentaires esclaves dès le départ, donc seulement un *start* de l'extérieur pourra les déverrouiller, via le FIFO, comme décrit à la section 4.1. Enfin, le « *main* » appelle la fonction « init » puis la fonction « start_engine » pour initialiser les différents processus élémentaires et lancer l'ordonnanceur.

4.5 Fonction « change_thread » et la macro Pause

La fonction « change_thread » (Tableau 4-2) permet de passer d'un contexte à un autre. Son implantation peut être vue dans le fichier i_chip.c (annexe p.104). Cette fonction incrémente l'identificateur courant de processus élémentaire et elle vérifie si le nouveau processus élémentaire adressé par cet identificateur peut être lancé. Si le processus élémentaire est verrouillé, la fonction continue d'incrémenter le compteur pour essayer de trouver un autre processus élémentaire dans la liste qui pourrait être lancé. Lorsque le pointeur de processus élémentaires est rendu à la fin de la liste, on fait alors une lecture de l'état du FIFO (io_polling), puis on recommence à parcourir les différents processus élémentaires.

Lorsqu'on choisit un processus élémentaire, on vérifie s'il a déjà été lancé une fois. Pour cela, on utilise la tableau « thread_started », qui indique ce cas. Si le processus élémentaire n'a jamais été lancé, le noyau en assembleur devient alors nécessaire. Il faut changer le pointeur de pile pour qu'il pointe sur le bon contexte. Ensuite, on peut lancer le processus élémentaire en appelant sa fonction, tel qu'indiqué dans le tableau « thread_func ». Si le processus élémentaire a déjà été lancé, on appelle plutôt la commande *longjmp* en utilisant le contexte sauvegardé auparavant. Pour comprendre l'utilité de cette instruction, il faut examiner le fonctionnement des commandes *setjmp* et *longjmp*. *Setjmp* permet de sauvegarder tous les registres du processeur. La commande

reçoit en paramètre un pointeur indiquant où conserver l'information en mémoire. La commande *longjmp* permet quant à elle de recharger les registres à partir des données conservées par le *setjmp*. Puisque le compteur d'adresse fait partie des registres conservés, on se retrouve à la ligne où le *setjmp* a eu lieu. Pour savoir si on vient d'effectuer un *setjmp* ou si l'on vient de recharger un contenu, il faut regarder la valeur de retour de *setjmp*. La fonction retourne zéro s'il s'agit d'une sauvegarde et un si le contexte vient d'être rétabli.

La macro « Pause » est utilisée pour sauvegarder le contexte du processus élémentaire. Elle est responsable d'appeler la fonction « *setjmp* ». En argument au « *setjmp* », on fournit un endroit où stocker le contexte, c'est-à-dire dans le tableau prévu à cette fin. Lorsqu'on appelle cette fonction, la valeur de retour est nulle, ce qui implique qu'on appelle la fonction « change_thread ». Par contre, lorsque la fonction « longjmp » est appelée, on revient exactement dans le « if », mais cette fois avec une valeur de retour non nulle, donc l'exécution se poursuit. C'est grâce à ce petit mécanisme que nous avons conçu la bibliothèque. De plus, il est intéressant de remarquer qu'il n'est plus nécessaire de manipuler le pointeur de pile avec le noyau en assembleur lors d'un longjmp, celui-ci étant inclus dans le contexte de sauvegarde.

4.6 Fonction io_polling

L'exécution du *io_polling*, consiste à aller lire à l'adresse où est branchée le FIFO de la Figure 4.1. Lorsqu'on lit cette adresse, on lit la donnée en tête du FIFO qui est implantée dans le *archi*. S'il n'y a pas de donnée, on lit la valeur zéro. Cette valeur est réservée et permet de la distinguer des identificateurs qui sont numérotés à partir de un. Sinon, on peut considérer qu'il s'agit de l'identificateur matériel qui désire communiquer un message. Le message peut être le début d'un *start* ou la fin d'un esclave. Dans un cas comme dans l'autre, on déverrouille le processus élémentaire associé à cet identificateur de communication en utilisant le tableau « ID_comm ». Enfin pour acquiescer la lecture

du FIFO, on réécrit l'identificateur lu dans celui-ci, ce qui permet d'éviter tout problème de synchronisme. Cette opération permet de dépiler la valeur du FIFO. Il suffit de regarder ensuite si d'autres valeurs sont présentes.

Reste à voir l'implantation des commandes de transfert de données. Tout d'abord, on constate que la commande « Start », à la Figure 4.3, a été remplacée par la commande « send » suivie de la commande « pause ». La commande « send » fait partie de l'API. Elle reçoit en paramètre l'identificateur de la communication. Son rôle est donc de verrouiller le processus élémentaire en cours, puis de placer le pointeur du processus élémentaire dans le tableau des communications (ID_comm) selon l'identificateur demandé. Elle doit également avertir le archi par un message. Pour cela, elle écrit l'identificateur à une adresse réservée, appelée « adresse de contrôle ».

Enfin, pour implanter les commandes *read*, *write* et *echo*, il suffit de manipuler directement les adresses qui ont été choisies par Picasso lors de la génération. On écrit ou on lit à ces zones d'adresses qui ne sont pas en mémoire, mais bel et bien stockées dans le *archi*.

4.7 Le bloc *archi*

Comme on l'a vue, le bloc *archi* est composé de blocs de base. La construction du *archi* dépend des spécifications fournies dans Picasso. Comme nous avons dit, le bloc *archi* agit comme un bus dédié entre le *Ichip* et les divers éléments matériels.

Il est à noter que pour fixer les adresses du FIFO, du contrôleur et des registres de données, nous avons proposé un mécanisme d'adressage assez simple. On doit tout d'abord déterminer le nombre de bits d'adresses disponibles et la plage d'adresse. Le *Ichip* nous fournit ces informations. Ensuite, nous considérons que les deux bits les plus significatifs contiendront le premier destinataire de la donnée. Il s'agit d'un décodage

rapide qui implique peu de matériel. Si la combinaison binaire est « 10 », alors on s'adresse au FIFO. Si c'est « 01 » alors on s'adresse plutôt au contrôleur. Dans tous les autres cas, on suppose que la donnée est destinée à un registre. Dans ce cas, on utilise les dix bits les moins significatifs. On considère dix bits comme suffisants. Puisque les données sont implantées par des registres, cela permet d'accéder 2^{10} registres de 32 bits, soit 32768 registres de 1 bit. Comme plusieurs bits d'adresses ne sont pas utilisés, on pourrait facilement en ajouter d'autres.

Le premier bloc que nous allons voir est le décodeur de destinataire. Il est connecté sur le *Ichip*. C'est cet objet qui reçoit les messages du *Ichip*, les interprète et agit en conséquence. Les messages prennent donc la forme de données et d'adresses. Deux adresses sont réservées pour les messages de contrôle. La première adresse est pour le FIFO, la seconde pour le contrôleur. Le reste des adresses détermine un destinataire parmi les différents registres de données.

La tâche du décodeur de destinataire est donc de recevoir les différentes informations et les aiguiller de la bonne façon. Il s'agit donc d'un simple processus combinatoire. En utilisant le protocole d'adresse expliqué lors du processus de génération, on est en mesure d'envoyer ou de recevoir des données du bon bloc matériel. Chaque bloc matériel possède une ligne d'activation (_select) qui est activée seulement lorsque l'entrée cs_bar est activée. De plus, en utilisant les informations de l'entrée read_write, on peut aiguiller correctement le sens des données. En synthèse, on obtiendra donc des tampons à 3 états.

On retrouve ensuite le FIFO. On a déjà dit que ce FIFO permettait aux modules matériels de communiquer avec le logiciel. À chaque coup d'horloge, il vérifie s'il est sélectionné par le décodeur de destinataire. Cela implique que le *Ichip* essaie de lire son contenu. Si ce n'est pas le cas, le FIFO vérifie un à un chacun des périphériques. Pour un périphérique en particulier, il vérifie si celui-ci veut être ajouté dans la queue. Dans ce

cas, l'insertion se fait et un signal d'acquittement est envoyé au périphérique. Comme l'horloge est assez rapide, on peut supposer qu'un périphérique attendra très peu avant d'être servi. La valeur qu'on insère dans le FIFO est le numéro d'identificateur. Cela permet au logiciel de retracer chaque appel provenant des systèmes matériels.

Si le FIFO est sélectionné par le *Ichip* au coup d'horloge, on doit regarder quelle opération est en cours sur le bus d'adresse. S'il s'agit d'une lecture, on place la valeur de l'élément de tête du FIFO sur le bus de données. Dans le cas d'une écriture, on compare la valeur sur le bus, avec la valeur de l'élément de tête. S'il y a égalité, on retire l'élément du FIFO. Puisque chaque donnée ne peut se retrouver qu'une seule fois dans le FIFO, cela permet d'éviter le problème où l'on retirerait trop de valeurs du FIFO sans les lire, à cause d'un signal qui resterait levé. En effet, un processus ne peut se réinscrire dans le FIFO, car il demeure en attente du logiciel, donc d'un message de contrôle envoyé sur une autre adresse. Puisqu'on ne peut retirer que la valeur en cours, ce petit mécanisme permet d'éviter les situations semblables à des verrous mortels (*deadlocks*).

Le bloc suivant à regarder est le contrôleur. Le contrôleur gère l'état de fonctionnement ou d'arrêt des différents modules matériels. Lorsqu'il est appelé, il regarde les bits les moins significatifs du bus de données. Cela donne l'identificateur du bloc qui doit recevoir le message de contrôle. Le contrôleur peut donc déterminer la bonne interface à avertir. L'avertissement est transmis le long d'un signal appelé « copro_resume ». Ce signal, d'une durée d'une période d'horloge, permet à l'interface concernée de changer son état interne.

Le prochain module à examiner est le décodeur de registre. Il s'agit d'un module combinatoire. Ce module se comporte comme un multiplexeur dont la ligne de sélection serait l'adresse. L'adresse permet de sélectionner le bon registre matériel, donc le bon port. Le bus de données se connecte à ce registre ce qui permet de le lire ou d'y écrire. Encore une fois, il faut analyser le sens des données, ce qui fait que la synthèse produira

des tampons à 3 états. Lorsque nous sommes en lecture, l'adresse détermine quel registre doit être accédé. Sa valeur est envoyée sur le bus de données. Dans le cas d'une écriture, on envoie la valeur du bus de données vers le bon registre.

Maintenant, on peut regarder les blocs d'interfaces matériel maîtres et esclaves (Figure 4.1). Commençons par les maîtres. Ils sont décrits sous forme d'une machine à 4 états séquentiels. Ces quatre états sont illustrés dans le diagramme temporel à la Figure 4.4. L'interface commence par signaler qu'elle est libre, en mettant le signal « unlock » à '1'. Puis, elle attend que le signal d'entrée « start », provenant du bloc matériel, soit activé à '1'. À ce moment, elle demande à être inscrite dans le FIFO, puis attend la confirmation de celui-ci. Enfin, elle attend le signal « resume_copro » venant du contrôleur. Ce signal sera émis quand l'esclave en logiciel aura terminé sa tâche. À ce moment, elle pourra ramener le signal « unlock » à 0. Cela pourra libérer le bloc matériel. Enfin, il ne restera qu'à attendre que le « start » soit ramené à '0' pour être certain que la tâche est bien terminée. L'interface retourne ensuite à la première phase. C'est donc une mécanique assez simple et efficace.

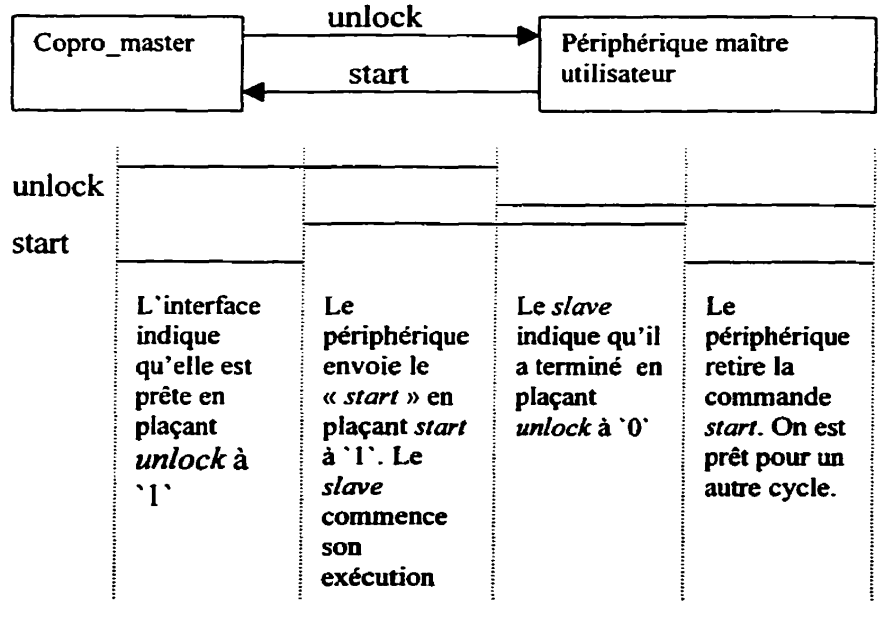

Figure 4.4 Protocole pour la commande start pour une interface maître

Le cas de l'interface esclave est naturellement similaire. Elle commence tout d'abord par attendre le signal provenant du contrôleur, soit « resume_copro ». Lorsque ce signal est détecté, elle vérifie si la ligne « unlock » de l'esclave branché à l'interface est disponible. Si c'est le cas, alors l'interface peut envoyer le signal « start ». Puis, elle attend que la ligne « unlock » provenant de l'esclave redevienne disponible. À ce moment, elle peut arrêter le signal « start », puis demander à être inséré dans le FIFO. Une fois la confirmation de l'insertion arrivée, l'interface retourne en attente du prochain signal du contrôleur.

Cela complète notre étude des différents modules du bloc « archi ». On a donc couvert complètement l'architecture et le fonctionnement du système généré. Dans le chapitre suivant, nous verrons comment Picasso peut transformer une spécification de haut niveau vers cette architecture.

Chapitre 5 – Algorithmes de transformation et génération

Dans ce chapitre, nous allons expliquer les algorithmes qui transforment les spécifications haut niveau vers l'architecture présentée. Il s'agit donc d'obtenir des fichiers C et VHDL standards ainsi que les fichiers de configuration pour simuler avec l'outil Seamless ™. Nous allons voir les différentes techniques que nous avons conçues et implantées dans le logiciel Picasso. Il s'agit d'une section contenant beaucoup de détails et donc qui vise davantage un public intéressé à implanter ces techniques. Il s'agit d'un ensemble d'étapes qui transforme graduellement la spécification en états intermédiaires jusqu'au système final.

Les principales étapes sont les suivantes :

1. Mise à niveau hiérarchique
2. Regroupement du code logiciel par instance de processeur
3. Communication logicielle sur un même processeur
4. Implantation des types de données
5. Raffinement des modules matériels
6. Génération des modules *Ichip*
7. Génération des décodeurs de registres des *Archi*
8. Génération des modules *Archi*

La spécification de haut niveau peut souvent contenir des modules hiérarchiques. Il s'agit de modules contenant une description de l'interconnexion en sous-modules. La mise à niveau hiérarchique consiste donc à se débarrasser de ces modules hiérarchiques en aplatissant l'ensemble de la structure. Ensuite, puisque chaque bloc logiciel est relié à une instance de microprocesseur, on regroupe ensemble les spécifications logicielles qui vont sur les mêmes instances. On appelle cette étape l'inclusion. On se retrouve donc avec le même nombre de blocs logiciels que d'instances.

Suite à ce regroupement, il est possible que deux modules logiciels qui communiquaient se retrouvent sur un même processeur. On peut donc ajuster le code en conséquence pour implanter une communication logicielle/logicielle.

Pour être capable d'implanter les communications logicielles/matérielles, on doit traduire les types de haut niveau en C et VHDL équivalent. Ces types feront partie du raffinement des blocs logiciels et matériels. En raffinant les modules matériels vers l'architecture de communication présentée précédemment, on élimine toutes les extensions ajoutées au C et au VHDL (chapitre 3) pour réaliser la communication. On pourra ensuite traduire les blocs logiciels en module *Ichip* et *archi* et ainsi obtenir le système final.

5.1 Mise à niveau hiérarchique

La première transformation consiste à briser la hiérarchie (*flattening*). C'est donc dire qu'on veut retirer toutes les boîtes hiérarchiques du système et les remplacer par leur contenu. Cela va nous donner une nouvelle représentation contenant seulement des objets logiciels et matériels, au lieu d'avoir plusieurs niveaux hiérarchiques à synchroniser par la suite. Les objets hiérarchiques seront donc tous remplacés à cette étape. Il s'agit d'une mise à niveau structurelle et non pas fonctionnelle. C'est-à-dire que l'on refait l'ensemble des connexions entre les boîtes, mais pas les spécifications à l'intérieur de celles-ci.

Pour réaliser cette étape, nous employons un algorithme qui parcourt la liste de tous les objets présents dans le document à générer. Nous utilisons également une structure de donnée particulière que nous appelons un dictionnaire. Il s'agit en fait que d'une table qui associe le nom d'un objet ainsi que sa position dans la hiérarchie à un nouveau nom. Lorsqu'on brise la hiérarchie, on crée au niveau en cours des nouvelles instances des objets contenus dans les sous-niveaux symbolisés par l'objet hiérarchique. Des conflits de noms peuvent survenir, c'est pourquoi on doit les changer et maintenir

une table de correspondance entre les anciens noms (et leur chemin) et les nouveaux noms.

Pour chaque objet, on vérifie s'il s'agit d'un symbole ou d'un port. Pour un symbole, on procède tout d'abord à une validation. On vérifie si le symbole correspond bien à l'interface en mémoire du même symbole. La représentation en mémoire est la version la plus récente du symbole. Si le symbole n'a pas été rafraîchi, il se peut qu'il corresponde à une ancienne version, il faudra donc prévenir l'usager. Si l'objet est valide, on peut poursuivre en l'enregistrant dans le dictionnaire. S'il s'agit d'un port plutôt que d'un symbole, la validation n'est pas requise et donc on peut l'entrer directement dans le dictionnaire.

On peut ensuite balayer tous les objets du document, à la recherche d'objets hiérarchiques. Chaque fois qu'on en trouve un, on l'ouvre et on crée les instances qu'il contient dans le niveau en cours. On procède aux mêmes validations pour les symboles et on enregistre les nouvelles instances dans le dictionnaire. À la fin de cette étape, on s'occupe de déplacer les liens de communication. Pour chaque port de l'objet hiérarchique, on vérifie s'il y a des connexions présentes. S'il n'y en a pas, on recherche le port équivalent qui vient d'être créé et on l'efface. Si on trouve un lien, on peut trouver ensuite le port de destination. Ce port deviendra la nouvelle destination pour tous les liens branchés sur le port hiérarchique. Il faut rechercher maintenant tous les liens pointant sur l'ancien port hiérarchique, et les faire pointer sur le nouveau port de destination. Lorsqu'on a terminé, on peut effacer l'objet hiérarchique de départ. Lorsqu'il ne reste plus d'objets hiérarchiques, on a terminé cette partie. La Figure 5.1 présente une application de l'algorithme. La figure 5.1a contient un objet hiérarchique appelé « hier ». L'objet est défini dans la figure 5.1b. Dans la figure 5.1c, on voit le résultat après que l'algorithme ait brisé la hiérarchie. L'objet « hier » ne fait plus partie de la description.

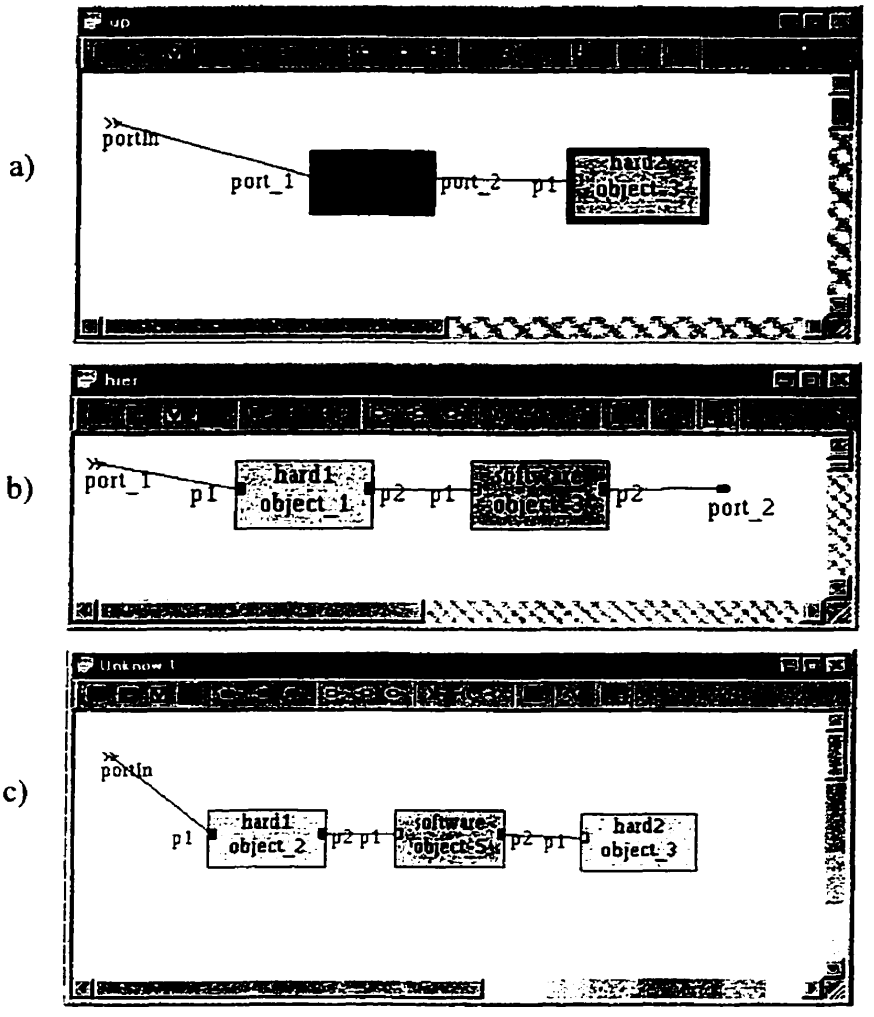

Figure 5.1 Exemple d'une mise à niveau

Dans l'avenir, le fonctionnement de cette partie devrait être raffiné. Au lieu de briser la hiérarchie, elle devrait plutôt faire partie intégrante du processus de raffinement. Dans le cas de systèmes complexes, c'est souvent à l'étape du « flattening » que la fiabilité de l'outil est mise en jeu. La complexité du système pousse l'outil à sa limite. De plus, il serait intéressant de conserver les informations relatives à la hiérarchie pour faciliter le déverminage du système final. La partie matérielle pourrait être hiérarchisée au niveau du VHDL en introduisant des couches successives de VHDL structurelles. Le problème vient du côté logiciel; le C est toujours à un seul niveau de fichiers. Une

amélioration serait de fournir au simulateur C un outil de navigation à travers les différents fichiers, de type arborescent (comme dans Microsoft Visual C++™). Les données sur l'arborescence seraient fournies par Picasso. L'usager pourrait donc se retrouver plus facilement dans le système final.

5.2 Regroupement du code logiciel par instance de processeur

La prochaine étape consiste à regrouper les blocs logiciels entre eux selon les instances de processeur définies par l'usager. Cela permet de réduire le nombre de blocs logiciels. On se retrouve avec un bloc logiciel pour chaque instance définie. Il faut donc commencer par vérifier si tous les blocs logiciels sont associés à une instance de processeur. Si ce n'est pas le cas, on doit prévenir l'usager.

Pour réaliser cette étape, l'algorithme procède à une analyse syntaxique des différents modules logiciels. On recrée un nouveau fichier de type logiciel, avec une section *port* et des sections *thread*, *slave* et *init*. La section *port* provient de la concaténation des informations contenues dans chaque section *port* des blocs logiciels. Tous les ports de tous les blocs logiciels sont donc rassemblés. Pour éviter les répétitions de noms, on préfixe le nom des ports par le nom de son module. La même chose est effectuée pour les blocs d'initialisation.

Les différents processus élémentaires et processus élémentaires esclaves (*slave)* sont copiés dans le nouvel objet logiciel. Chaque fois, on préfixe leur nom par le nom de l'objet pour s'assurer de l'unicité. Au sein même de chaque processus élémentaire, on doit effectuer certaines modifications. Tout d'abord les variables globales sont renommées comme on l'a mentionné au chapitre 3 section 3.3. Les variables globales sont préfixées par le mot « GLOBAL ». Il suffit donc de remplacer ce préfixe par le nom de l'objet.

On procède de la même façon pour les ports. Chaque port, avant d'être recopié, doit être renommé en le préfixant du nom de l'objet. On doit également rechercher chaque fonction *Start*, *Read*, *Write* et *Echo* pour corriger le nom du port sur lequel portent ces fonctions, pour que le tout concorde bien ensemble.

Une fois cette opération complétée, on peut créer une instance du nouvel objet dans le système. Il faut ensuite déplacer tous les liens connectés sur les anciens blocs logiciels vers le nouveau. On peut effacer par la suite les anciens blocs. On se retrouve donc avec un seul bloc logiciel pour chaque instance de processeur utilisée.

Pour facilité le déverminage du système, il faudrait songer à développer une façon de garder un lien entre le design d'origine et le système généré. Par exemple, on pourrait utiliser une technique semblable au « back annotation », bien connue dans la méthodologie de conceptions matériels. En effet, il se pourrait pour différentes raisons, que le système généré ne soit pas la représentation parfaite de la spécification de l'utilisateur. Cette erreur pourrait être reliée à la spécification de l'usager, ou simplement à une erreur du module de génération. Ainsi, un système de « back annotation » viendrait simplifier la tache de recherche à travers le code généré, pour y retrouver l'erreur et sa source.

Une amélioration intéressante serait une syntaxe qui viendrait préciser les caractéristiques des différentes fonctions d'entrées/sorties effectuées. Cela pourrait être le travail d'un outil d'estimation par exemple. Ainsi on faciliterait la tâche de l'algorithme dans le choix d'un mécanisme de communication, s'il y en a plusieurs.

Une autre amélioration serait d'éviter le recopiage inutile de code. Si on crée deux instances ou plus d'un même type d'objet logiciel, et qu'on les associe à la même instance de processeur, on se retrouve avec le même code copié plusieurs fois sous des noms de processus élémentaires différents. Il faudrait donc vérifier cette situation et la

corriger. Une correction facile serait de considérer que chaque processus élémentaire possède une fonction d'entrée. Puis il faudrait associer à la liste des processus élémentaires actifs, sa fonction d'entrée. On aurait donc deux processus élémentaires ou plus ayant la capacité de démarrer avec la même fonction, mais puisque chaque processus élémentaire possède sa propre pile, les variables seraient bien séparées.

5.3 Communication logicielle sur un même processeur

La prochaine étape consiste à procéder à l'inclusion du code. Après avoir rassemblé les blocs logiciels sur un même processeur, il est possible qu'un processus élémentaire se retrouve avec son processus élémentaire esclave. C'est donc dire qu'un lien de communication part d'un port maître du symbole et revient sur un port esclave du même symbole. Puisque le mécanisme de communication est bloquant, on peut remplacer le processus élémentaire esclave par une fonction. La commande *Start* devient donc un appel à cette fonction. En effet, en appelant la fonction, le maître bloque, en ce sens qu'il est en train d'exécuter la fonction du processus élémentaire esclave, pas son propre code. Quand la fonction est terminée, il revient dans son code. On constate donc que le comportement est vérifié, ce qui justifie la modification. Pour ce qui est des données, on peut les placer dans des variables globales. Puisque le maître et le processus élémentaire esclave ne les accéderont pas en même temps, cela ne cause pas de problème. Ainsi, les commandes *Read*, *Write* et *Echo* porteront sur ces variables. Le maître et l'esclave pourront donc les accéder.

La réalisation de cette étape est très simple. On peut séparer les modifications effectuées pour chaque lien en trois étapes. La première consiste à modifier la section *port*, la seconde à modifier la partie maître (*master)* et enfin la troisième, à modifier la partie esclave(*slave*). Pour les ports, on les remplace par des variables globales. Dans la partie maître, on recherche tous les appels aux commandes *Read*, *Write*, *Echo* et *Start*. Pour les trois premières, on modifie la commande pour qu'elle adresse la variable globale

et non le port. Pour le *Start*, on modifie la commande par un appel de fonction tout simplement. Dans le cas de l'esclave, on procède de la même façon pour les commandes *Read*, *Write* et *Echo*. Il faut noter cependant que les variables globales sur lesquelles portent ces commandes sont inversées. Si on fait un *Read* sur une variable, on retrouve la même variable avec un *Write* dans l'esclave.

5.4 Implantation des types de données

La prochaine étape consiste à traduire les types abstraits en descriptions accessibles pour chaque langage utilisé, actuellement le C et le VHDL. Cela va permettre d'obtenir une implantation concrète des données qui circulent dans le système.

Chaque type comme « PIC_int », « PIC_struct », ... est défini par une classe. Toutes ces classes de type héritent du même parent, y compris les classes complexes qui décrivent les structures et les tableaux. Chaque fois qu'un nouveau type de donnée est créé, il s'enregistre dans une liste de tous les types du système. Pour construire la représentation dans un langage donné, il suffit de parcourir cette liste et de demander à chaque objet de se générer. Chaque objet dispose actuellement de deux fonctions membres appelées : « GenerateC » et «GenerateVHDL ». Puisque ces fonctions sont virtuelles, chaque objet sait comment implanter son type dans le langage approprié. Pour les types complexes, on peut construire un canevas de base et appeler de façon récursive les mêmes fonctions pour les éléments constitutifs du type.

Une contrainte a été introduite dans l'élaboration de l'outil. La représentation matérielle des types a tout d'abord été fixée, puis des équivalents logiciels ont été trouvés. En partie, cela vient du fait que l'implantation logicielle des types dépend du type de microprocesseurs utilisés. Pour un entier par exemple, il faut savoir si les données sont en format petit-boutiste (en anglais, *little endian*) ou gros-boutiste (en anglais, *big endian*). Également, il faut connaître le comportement du compilateur. Par exemple, il

faut savoir comment sont alignés les membres d'une structure en mémoire (surtout si la grandeur de chaque membre est différente). Ces informations font parties de l'instance du processeur et sont utilisées par la fonction « generateC ».

5.5 Raffinement des modules matériels

L'étape suivante consiste à implanter les extensions présentes dans les fichiers VHDL pour la communication et la synchronisation. Cette étape permet de transformer les commandes telles que *Read*, *Write*, etc. par des énoncés VHDL implantant ces fonctionnalités. Également, tous les ports de haut niveau sont ramenés à un niveau VHDL. À la fin de cette étape, chaque objet VHDL ne contient que des ports et des commandes VHDL standard.

Pour réaliser cette étape, on commence tout d'abord par créer un nouveau fichier VHDL qui contient les en-têtes des bibliothèques et des paquets à utiliser. Il faut inclure les bibliothèques IEEE et aussi le paquet de définitions de types créé à l'étape 5.4. Ensuite il faut recopier les ports inclus dans l'entité, en prenant soin de remplacer les ports maîtres et esclaves par des ports VHDL. Si le port est une entrée maître ou esclave, on remplace celui-ci par un bus de données de type « in » en VHDL. Dans le cas d'une sortie maître ou esclave, on remplace plutôt par un bus de type « inout » en VHDL. Cela permet de lire une donnée déjà envoyée. Donc, pour les ports maîtres ou esclaves de type « inout » selon Picasso, on a deux bus VHDL qui sont créés, un « in » et un « inout ». Pour éviter des conflits de noms, on ajoute un suffixe au nom des bus, soit « data_in » et « data_out ».

Pour contrôler adéquatement le transfert de données des ports maîtres/esclaves, on ajoute des ports appelés *start* et *unlock*. Ces ports permettent d'implanter le protocole de communication pour la commande *Start*. Nous avons présenté au chapitre précédant le fonctionnement du protocole (section 4.7).

Cela complète le raffinement de l'entité. On doit ensuite trouver et analyser chaque *process* et chaque esclave présent dans l'architecture. Dans les deux cas, le comportement est à peu près le même. Dans le cas d'un processus, on vérifie s'il possède l'énoncé *Comm_attrib* immédiatement après le *begin*. Si ce n'est pas le cas, on assume que le processus n'utilise pas du tout les extensions du langage et donc qu'il peut être recopié intégralement. Sinon, on recherche les différents éléments du langage et on les traite. Les énoncés *Read* et *Echo* sont remplacés par le port équivalent. Pour l'énoncé *Write*, on utilise l'opérateur d'affectation du VHDL (<=) pour envoyer la valeur sur le bon bus. La commande *Start* est un peu plus complexe. On doit implanter les énoncés du protocole en utilisant les valeurs du *Comm_attrib* comme signaux d'horloge et de remise à zéro.

Si au lieu d'un processus, nous avons un module matériel esclave, le traitement pour les commandes demeure le même. Ce qu'il y a de plus, c'est que le module matériel esclave doit être traduit sous la forme d'un processus. Il faut donc prévoir un mécanisme pour placer le processus en attente du *Start* dès le départ. Également, il faut prévoir un système pour avertir le processus maître que l'esclave a terminé son exécution. On s'attend donc à trouver un énoncé *Comm_attrib* pour ce genre de processus. La Figure 5.2 nous montre le code de l'additionneur du chapitre précédant ainsi que la version en VHDL standard.

```
entity adder_v is
port(
   value:slave_inout PIC_sum => PIC_int;
   clk:in std_logic;
   reset:in std_logic
);
end adder_v;

architecture picasso of adder_v is
begin

value:slave
variable i:PIC_int;
variable j:PIC_int;
variable k:PIC_int;
begin
            Comm_Attrib(clk.reset);
            i:=Read(value.x);
            j:=Read(value.y);
            Write(value.i+j);
end slave;

end picasso;
```

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_signed.ALL;
use work.data_type.all;

entity adder_v_final_vhdl is
port(
clk : in  std_logic;
reset : in  std_logic;
value_data_in : in PIC_sum:=(0,0);
value_data_out : inout PIC_int:=0;
value_start : in std_logic;
value_unlock : out std_logic
);
end adder_v_final_vhdl;

architecture picasso of adder_v_final_vhdl is
begin

value : process
variable i:PIC_int;
variable j:PIC_int;
variable k:PIC_int;
begin
 reset_loop: loop       --init
 value_unlock<='1';
 while value_start/='1' loop
   wait until clk'event and clk='1';
   if (reset='1') then exit reset_loop; end if;
end loop;
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;
            i:=(value_data_in.x);
            j:=(value_data_in.y);
            value_data_out<=i+j;
value_unlock<='0';
while value_start='1' loop
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;
end loop;
end loop reset_loop;
end process value;

end picasso;
```

Figure 5.2 Transformation de l'additionneur matériel par Picasso (avant & après)

Lorsqu'on a terminé de bâtir le nouveau code, on peut créer une instance du nouvel objet. Plus tard, on déplacera les liens attachés à l'ancien objet vers le nouveau, et on pourra ensuite détruire l'ancien objet.

Cette façon de procéder pourrait être améliorée. En effet, nous générons tout le matériel en une étape. Cela est possible, car on sait comment le logiciel va implanter les communications. Si plusieurs mécanismes de communication sont introduits, il faudrait alors procéder en deux étapes. L'approche serait semblable à la façon de faire des compilateurs. La première « passe » correspond à un raffinement du code C pour obtenir du code objet. Les éléments de communication non-résolus sont exprimés sous une forme symbolique et réduite. On répète cette étape pour chaque fichier. Puis, on procède à une deuxième étape, où on connecte les objets ensemble et où on remplace les liens symboliques par de vraies communications, en général, en appels de fonctions.

5.6 Génération des modules *Ichip*

L'étape suivante consiste à créer les instances de processeur (*Ichip)* et les blocs protocole (*archi)* associés au code logiciel. Le logiciel n'a pas d'existence réelle. Sous sa forme compilée, il se retrouve dans une mémoire attachée à un microprocesseur. Il faut donc créer les blocs qui lui permettront d'exister (*Ichip*) et aussi d'interagir (*archi*) avec le monde extérieur.

Pour réaliser cette étape, on utilise le générateur de *Ichip* propre à un microprocesseur donné. Ces générateurs sont utilisés par Picasso pour créer les composants. Le générateur reçoit le module en C en entrée et génère le module matériel le supportant. Tous les modules matériels possèdent un port de communication capable de communiquer avec un *archi*.

Le générateur commence par remplacer les extensions au C présentes dans le fichier pour un *Ichip* donné. La Figure 5.3 présente le client logiciel du chapitre 3 ainsi que sa version étendue. Le générateur fixe à l'interne un numéro d'identification (ID) pour tous les ports. Cette liste va nous donner les numéros d'identification pour les processus élémentaires esclaves. Par exemple, si le port #1 s'appelle « *value* », alors, on peut décider que « 1 » deviendra le numéro d'identification pour le processus élémentaire esclave « *value* ». Lorsque le matériel communiquera avec le *archi*, il utilisera ce numéro pour identifier le processus élémentaire.

```
port {
my_port : master_inout PIC_sum =>
PIC_int;
}


thread my_thread{
   PIC_int result;

   Write(my_port.x,40);
   Write(my_port.y,50);
   Start(my_port);
   result=Read(my_port);

   Pause;
}
```

```
/* generated by picasso, 1999, 2000 */
#include "data_type.h"
#include "i_chip.h"


#define object_2_my_port_data_in (*(PIC_int*)1073741824)
#define object_2_my_port_data_out (*(PIC_sum*)1073741828)
#define object_3_receptor_data_in (*(PIC_sum*)1073741836)
#define object_3_receptor_data_out (*(PIC_int*)1073741844)

void my_thread_thread()
{
 for(;;){
   PIC_int result;

   object_2_my_port_data_out.x=40;
   object_2_my_port_data_out.y=50;
   send(1);
   Pause;

   result=(object_2_my_port_data_in);
   Pause;
 }
}

void init() {}

Tthread_func thread_func[]={my_thread_thread};

void main()
{
        init_thread();

        /*lock the slave thread*/
        Lock(1);

        StartEngine();
}
```

Figure 5.3 Transformation du client logiciel par Picasso (avant & après)

Ensuite, le générateur doit procéder à une analyse des différentes zones principales du programme : *port*, *threads* et *slaves*. Ce qui n'appartient pas à ces zones n'est pas touché (fonctions, variables globales,...) Pour la zone *port*, on remplace chaque port par un pointeur qui correspond à une adresse sortante du système, c'est-à-dire une adresse qui n'est pas décodée par la mémoire du *Ichip* mais envoyée au *archi*. Chaque pointeur porte le nom du port avec comme suffixe « data_in » ou « data_out » dépendant du sens des données. Le *Ichip* réserve un intervalle d'adresses qui n'est pas décodé par la mémoire. Lorsqu'on utilise ces adresses, un signal (*CS*) active le port de communication du *Ichip* vers le *archi* et l'adresse est envoyée. On peut alors lire ou écrire une donnée sur le bus. Il s'agit du seul mécanisme de communication supporté en ce moment. Il possède des temps d'accès identiques à ceux d'une mémoire.

Il faut prendre soin dans l'allocation des ports de tenir compte des grandeurs des types que le compilateur utilisera. Si on revient à la Figure 5.3, le port de communication utilisé dans l'exemple (my_port : master_inout PIC_sum –> PIC_int) est décomposé en deux variables globales. On crée tout d'abord « my_port_data_in » de type « PIC_int » et « my_port_data_out » de type « PIC_sum ». Il est important de se rappeler comment le compilateur va aligner les éléments de la structure « PIC_sum ». Puisqu'il s'agit d'une structure de deux entiers, le membre appelé « .x » sera confondu avec l'adresse de l'élément. Par contre, pour l'adresse du membre « .y », on peut montrer qu'il s'agit de l'adresse de la structure augmentée de la taille de « .x », donc de 4 octets. Ces informations sont nécessaires pour construire le matériel permettant de décoder le destinataire d'une donnée.

Toujours selon la Figure 5.3, on doit ensuite s'occuper de chaque ligne faisant un appel à une commande *Read*, *Write*, *Echo* ou *Start*. Pour les trois premières commandes, on utilise le bon nom de pointeur. Pour la commande *Start*, on utilise la fonction *send* de l'API du I*chip*, avec le numéro d'identification de l'esclave à appeler. Cette fonction utilise l'adresse de contrôle pour envoyer l'identificateur du processus élémentaire au

archi. On ajoute également tout de suite après un énoncé *Pause*. Celui-ci permet d'arrêter le processus élémentaire et de passer le main à un autre comme à l'habitude. Par contre, l'ordonnanceur mémorise qu'un *send* est en cours et il ne redonnera pas la main à ce processus élémentaire tant que le *send* ne sera pas complété.

Finalement, le générateur se met à la recherche de chaque bloc *thread* et *thread* esclave. Chaque fois qu'un processus élémentaire est trouvé, il est transformé en une fonction normale. À l'intérieur de celle-ci, on encadre le code dans une boucle infinie. Il devient donc impossible de sortir de cette boucle. On ne peut que le suspendre en utilisant la commande *Pause*. Dans le cas d'un processus élémentaire esclave, on ajoute en plus des instructions avant la fin de la boucle indiquant qu'il a terminé son exécution. Pour cela, on effectue encore un appel à la fonction *send* avec l'identificateur du processus élémentaire esclave. Puis, on fait une pause pour donner la main à un autre processus élémentaire. Là encore, le *send* va placer le processus élémentaire esclave en attente d'un prochain *Start*.

5.7 Génération des décodeurs de registres des *archi*

L'étape suivante consiste à générer le décodeur de registres. Celui-ci fait partie du *archi*. Comme on l'a déjà vu au chapitre 4, section 4.7, ce bloc est responsable de la traduction des adresses vers les registres. Il s'agit en fait d'un multiplexeur bidirectionnel couplé avec un contrôleur. Celui-ci aiguille le bus de données sur le bon registre en fonction de l'adresse fournie. Il doit également veiller à ce que le sens du bus soit correct, dépendant si on est en mode lecture ou écriture.

Pour créer ce composant, il faut utiliser les adresses des éléments de données, leur taille et leur alignement en mémoire. On doit décomposer chaque structure complexe en ses parties élémentaires. Puis, on doit calculer l'adresse de chacun de ses éléments et bâtir une table permettant ces accès. Nous avons développé pour cela un algorithme récursif

qui balaie les classes de types. Puisque chaque type est une entité héritant du même parent, on peut bénéficier des avantages du polymorphisme. On peut demander à un élément complexe (comme une structure ou un tableau) combien d'éléments élémentaires le composent. On peut ensuite lui demander de retourner chacun de ses éléments et leur taille. Connaissant l'adresse de base, déjà fixée, et la taille de chaque membre, il devient possible de construire la table. Pour l'instant, si on reprend l'exemple de la structure « PIC_Sum », on a deux éléments primaires. Le premier retourne comme chaîne de caractère « .x » et zéro comme offset. L'autre « .y » et quatre comme décalage, car on a supposé que x était un entier qui loge sur quatre octets

Puisque l'algorithme est récursif, on peut mélanger des tableaux et des structures ensemble, et l'algorithme va tout décomposer en éléments élémentaires, comme le ferait un compilateur.

Enfin, lorsque cette étape est complétée, il reste à générer le script de compilation. Il suffit d'y inclure les commandes appelant le compilateur avec les bons fichiers que l'on vient de créer.

5.8 Génération du module *archi*

Maintenant, il reste la génération de l'objet *archi*. On dispose d'un générateur qui s'occupe d'implanter la communication selon un modèle donné. Puisque actuellement on ne dispose que d'un seul mécanisme de communication, on n'a qu'un seul type de *archi*.

Le *archi* est un module très facile à paramétrer. Il a été conçu dans ce but. Également, le code qui le compose a été conçu pour que la synthèse ne produise pas des structures inutiles (comme des « *latchs* » superflues).

La première partie qui est générée est le bloc décodeur d'adresse du *archi*. Sa construction repose sur un ensemble de paramètres construit à partir d'informations obtenues du *Ichip*. La description contient entre autres la taille des différents bus (adresses et données), le nombre de périphériques, etc.

On peut ensuite créer le *archi* proprement dit. Le *archi* peut être décomposé en un sous-ensemble de blocs connectés entre eux. Le module de génération s'occupe de créer les bonnes instances des bons types de blocs et d'établir les connexions entre eux. Puis, il génère un fichier VHDL structurel décrivant ces connexions, avec les bons noms de port. L'utilité et le détail ayant déjà été vus, la construction de ce fichier relève donc simplement d'une analyse des spécifications et du respect des règles de VHDL.

Par la suite, on ajoute au fichier de compilation (*script*), les noms des fichiers VHDL du *Ichip* et du *archi*. Puis, on doit générer les fichiers de simulation pour l'outil de co-simulation Seamless. Pour ce faire, Picasso demande à chaque *Ichip* de fournir les informations nécessaires à sa construction. Il s'agit de transmettre le modèle de microprocesseur utilisé ainsi que des mémoires. En combinant ces résultats pour chaque *Ichip*, on peut construire le fichier souhaité. Enfin, on peut établir les connexions VHDL entre le I*chip* et le *archi*.

Il reste finalement à déplacer les liens entre les blocs. Il faut tout d'abord les balayer un à un. Pour chaque lien, on doit vérifier s'il connecte des ports de type VHDL ou de type maîtres/esclaves. S'il s'agit de ports VHDL, on identifie l'ancienne et la nouvelle version des objets à chaque bout du lien, et on peut déplacer le lien. S'il s'agit de ports complexes, on doit encore trouver l'ancienne et la nouvelle version de chaque objet connecté. Mais ici, au lieu de déplacer le lien, on connecte les signaux de contrôle et de données correctement. C'est-à-dire qu'on connecte les signaux *start* et *unlock* ensemble et le ou les bus de données. Enfin, on peut retirer les anciennes versions d'objets.

On obtient donc finalement un système qui contient seulement des blocs matériels. La dernière étape consiste à produire le fichier structurel qui contient les instances de chacun des blocs. Pour cela, un autre algorithme a été développé. La première phase de l'algorithme consiste à construire des groupes de liens. Il s'agit de construire la liste des liens qui partagent au moins un port entre eux. Puisque à ce niveau, il n'existe que des liens VHDL et que chaque lien peut être vu comme une opération d'égalité entre deux ports, on a affaire à un ensemble de liens transitifs (si A est branché à B et que B est branché sur C, quand la valeur de A change, l'entrée C va changer également). Chacun de ces groupes de liens correspond à un signal en VHDL. Puis, il faut parcourir toutes les instances présentes dans le système. Dans un tableau, on note le type de chacune d'entre elles, de façon à ne pas noter le type plus d'une fois.

On peut donc créer les différentes parties du fichier VHDL. L'entité ne contient que les ports VHDL sortants accompagnés de leur type. Ensuite, sachant qu'elles sont les types de boîtes présents dans le système, on peut créer les déclarations des composants. Puis ensuite. on crée les différents signaux utilisés pour connecter les groupes de liens entre eux. Enfin, pour chaque objet, on crée une instance et on la branche correctement aux différents signaux.

Cela complète donc la construction du système à partir des spécifications de haut niveau vers notre architecture type. Comme on l'a souligné, il y a de nombreux endroits dans le code où nous pourrions apporter des améliorations. Des améliorations plus profondes sont possibles également pour chaque étape présentée et sur leur imbrication. L'introduction de différents mécanismes de communication en est un exemple.

Le chapitre suivant présentera différents exemples utilisant la méthodologie proposée par Picasso

Chapitre 6 – Applications pratiques

Dans ce chapitre, nous présenterons quelques applications réalisées avec l'outil Picasso. Nous commencerons par voir comment un FIFO peut être réalisé afin de permettre à deux processeurs de communiquer ensemble et de se synchroniser. Cet exemple est inspiré de la documentation de SystemC [4] et permet de comparer cette description à la notre.

Nous verrons ensuite l'exemple complet de l'additionneur, présenté au chapitre 3. Cet exemple montre l'exploration de différentes implantations logicielles/matérielles et le gain apporté par Picasso.

Enfin, le dernier exemple qui sera vu est un petit routeur. Cet exemple illustre comment des composants déjà existants peuvent être réutilisés. On y voit un exemple de conversion d'un protocole maison vers l'environnement Picasso.

6.1 Le FIFO

La documentation du SystemC propose la mise au point d'un FIFO pour définir la relation typique de producteur/consommateur. La modélisation du FIFO utilise des RPC qui proviennent de l'environnement de Coware [32]. En fait, les concepts promulgués par Coware se retrouvent dans SystemC. À la Figure 6.1 nous présentons un extrait du manuel d'usager de SystemC qui donne la modélisation du FIFO. Pour aider à comprendre l'exemple, voici quelques points de soutien :

- SystemC est un ensemble de classes C++. Ces classes permettent la modélisation de systèmes embarqués. Tous les objets commençant par « sc_ » sont donc des classes. Exemple : sc_module, sc_inslave, etc.
- L'unité première de SystemC est le module. Un module communique par des ports. Dans notre cas, 2 ports de type esclave sont utilisés. L'un permet de déposer des données dans le FIFO, l'autre, d'en retirer. L'avantage des esclaves est de bloquer leur maître tant qu'ils n'ont terminé. Ainsi, si le FIFO est plein (resp. vide), le port

d'entrée (resp. sortie) va être bloquant jusqu'à ce que des données soient retirés (resp. présentes).

- Le constructeur sc_ctor définit les fonctions reliées aux esclaves. Aussitôt qu'une donnée est écrite sur un port esclave, la fonction associée est appelée. Lorsque la fonction termine, le maître l'ayant appelé peut reprendre son exécution.
- La variable « buffer » est utilisée pour stocker les données. Dans notre cas, il s'agit d'entiers.

```
SC_MODULE(fifo)
{
// ports
sc_inslave<int> Pwrite; // slave port
sc_outslave<int> Pread; // slave port

//member variables
buffer<int> buf; // FIFO buffer
int item; // buffer item

SC_CTOR(fifo)
{
SC_SLAVE( blockingWrite, Pwrite);
SC_SLAVE( blockingRead, Pread);
}

// slave methods

void blockingWrite()
{
  if (buf.full() ){
      do {wait();} // wait for sensitive edge of the producer clock
      while( buf.full());
  }
  // buffer is not full
  // write into buffer
  item = Pwrite; // read from slave port
  cout << "Writing into buffer: item = " << item <<endl;
  buf.put(item);
}

void blockingRead()
{
  cout << "\nfifo:blockingRead called" << endl;
  if (buf.empty()){
       do {wait();}
       // wait for sensitive edge of the consumer clock
       while (buf.empty() );
  }
  // buffer is not empty
  // read from buffer
  item = buf.get();
  cout << " Item read = " << item << endl;
  Pread = item; // write to slave port
}

};
```

Figure 6.1 Modélisation d'un FIFO en SystemC

Nous avons décidé de vérifier comment ce FIFO pouvait être implanté dans notre environnement. Nous avons créé un système de test basé sur certaines hypothèses. Le FIFO est implanté en matériel et permet à deux modules logiciels d'échanger des données. Chaque module logiciel a son propre microprocesseur, on modélise donc un système multiprocesseur. Il s'agit donc d'un système fréquemment utilisé. La Figure 6.2 représente le système tel que défini dans Picasso.

Figure 6.2 Système utilisant le FIFO

Le type de données qui sont échangées est appelé « Pic_FIFO » et est équivalent à un entier (Pic_Int). Si on regarde notre implantation du FIFO en matériel, Figure 6.3, on constate qu'elle est assez similaire avec la Figure 6.1. Elle utilise du VHDL et nos extensions de communication.

```
entity fifo is
port(
  Pwrite:slave_in Pic_FIFO;
  Pread:slave_out Pic_FIFO;
);
end fifo;

architecture picasso of fifo is
  constant fifo_size:integer:=10;
  type t_buff is array(0 to fifo_size-1) of integer;
  signal buf:t_buff:=(others=>0);
  signal r_index,w_index:integer:=0;
  signal full,empty,clk,reset:std_logic;
begin

Pwrite : slave
variable item:Pic_FIFO;
variable w_index_var:integer:=0;
begin
  comm_attrib(clk,reset);
  -- wait for not full signal
  while full='1' loop
          Wait_one_Clock;
  end loop;
```

```
    -- buffer is not full
    -- write into buffer
    item := Read(Pwrite); -- read from slave port
    buf(w_index_var)<=item;
    w_index_var:=w_index_var+1;
    if w_index_var=fifo_size then w_index_var:=0; end if;
    w_index<=w_index_var;
  end slave;

  Pread : slave
  variable item:Pic_FIFO;
  variable r_index_var:integer:=0;
  begin
    comm_attrib(clk,reset);
    -- wait for not empty signal
    while empty='1' loop
              Wait_one_Clock;
    end loop;

    -- buffer is not empty
    -- read from buffer
    item := buf(r_index_var);
    Write(Pread,item);
    r_index_var:=r_index_var+1;
    if r_index_var=fifo_size then r_index_var:=0; end if;
    r_index<=r_index_var;
  end slave;

  process(r_index,w_index)
  variable t:integer;
  begin
    full<='0';
    empty<='0';
    if r_index=w_index then
              empty<='1';
    end if;
    t:=w_index+1;
    if t=fifo_size then t:=0; end if;
    if t=r_index then
              full<='1';
    end if;
  end process;

  reset<='1', '0' after 30 ns;

  process
  begin
  clk<='0'; wait for 100 ns;
  clk<='1'; wait for 100 ns;
  end process;

  end picasso;
```

Figure 6.3 Implantation du FIFO dans Picasso

Les deux premiers processus sont équivalents aux deux processus esclaves. L'un s'occupe du port de lecture et l'autre du port d'écriture. Le troisième processus est en VHDL standard et n'utilise pas les extensions de communication. Il permet la gestion des signaux *full* et *empty* à partir de l'état du FIFO. Enfin, on retrouve les énoncés permettant de fixer les signaux d'horloges et de remise à zéro.

Le producteur et le consommateur sont implantés comme à la Figure 6.4. Le producteur commence par entrer dans une boucle d'attente. Cela créera une latence et placera le consommateur en situation de blocage. Puis, le producteur se met à envoyer des données et à les valider par la commande *start*. Le *start* va donc être bloquant si le FIFO est plein. Dans le cas du consommateur, il procède toujours à une lecture de donnée en utilisant la commande *start*. Cela va donc le bloquer si le FIFO est vide. Puis la commande *Read* permet de lire la donnée envoyée par le FIFO. Une boucle ralentit le consommateur, ce qui devrait créer une situation à long terme où le FIFO est toujours rempli et où le producteur est bloqué et en attente du consommateur.

```
port {
  Pout: master_out Pic_FIFO;
}

thread{
  Pic_FIFO val=0;
  int i,j;

  j=0; /*dummy loop to slow down the
initialization of the producer*/
        for (i=0;i<100;i++){
          j=j+i;
        }
  while (true)
  {
        val += 2;
        Write(Pout,val);
        Start(Pout);

  }
}
```

```
port {
  Cin: master_in Pic_FIFO;
}

thread{
        Pic_FIFO x=0;
  int i,j;
  while (true)
  {
        Start(Cin);
        x = Read(Cin);

        j=0; /*dummy loop to slow down the
consumer*/
        for (i=0;i<100;i++){
          j=j+i;
        }
  }
}
```

Figure 6.4 Implantation du producteur (à gauche) et du consommateur (à droite)

À partir de toutes ces informations, Picasso peut générer le système final. Il sera composé de deux *Ichip* et deux *archi* (chapitre 4). Les deux *archi* seront branchés au FIFO. Le système généré peut donc être simulé avec Seamless. Cet environnement est présenté à la Figure 6.5. On y retrouve le simulateur VHDL Modelsim et sa fenêtre de trace, la console Seamless et les simulateurs logiciels. Il y a un simulateur pour chaque processeur. À l'intérieur de ceux-ci on retrouve le code généré correspondant au producteur et au consommateur. Seamless permet donc de synchroniser ces simulateurs logiciels avec le simulateur matériel.

À l'aide des simulateurs logiciels, on peut vérifier si les données écrites correspondent aux données lues, ce qui est bien le cas. Si on simule le système suffisamment longtemps, on obtient la trace de la Figure 6.6. Deux points importants ont été soulignés sur la trace. Au point 1, on voit que le signal *start* qui contrôle le port de lecture est prolongé jusqu'à la première écriture. Cela permet de tenir le processus élémentaire correspondant en attente. Au point 2 on note que c'est le port d'écriture qui retrouve son signal *start* prolongé. En effet, le FIFO est plein, il bloque donc le processus

élémentaire correspondant. Les autres signaux du FIFO montrent son bon fonctionnement.

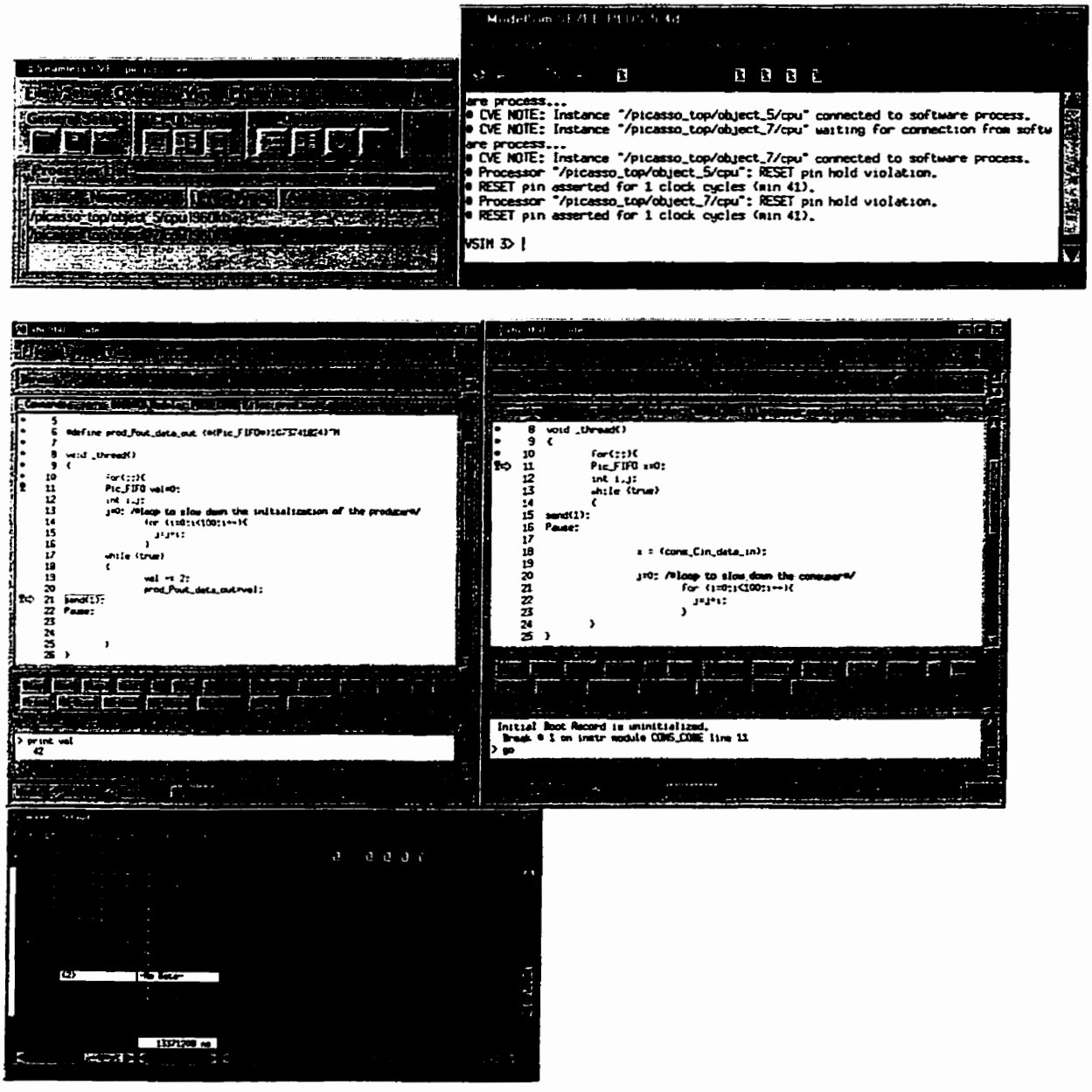

Figure 6.5 Environnement Seamless

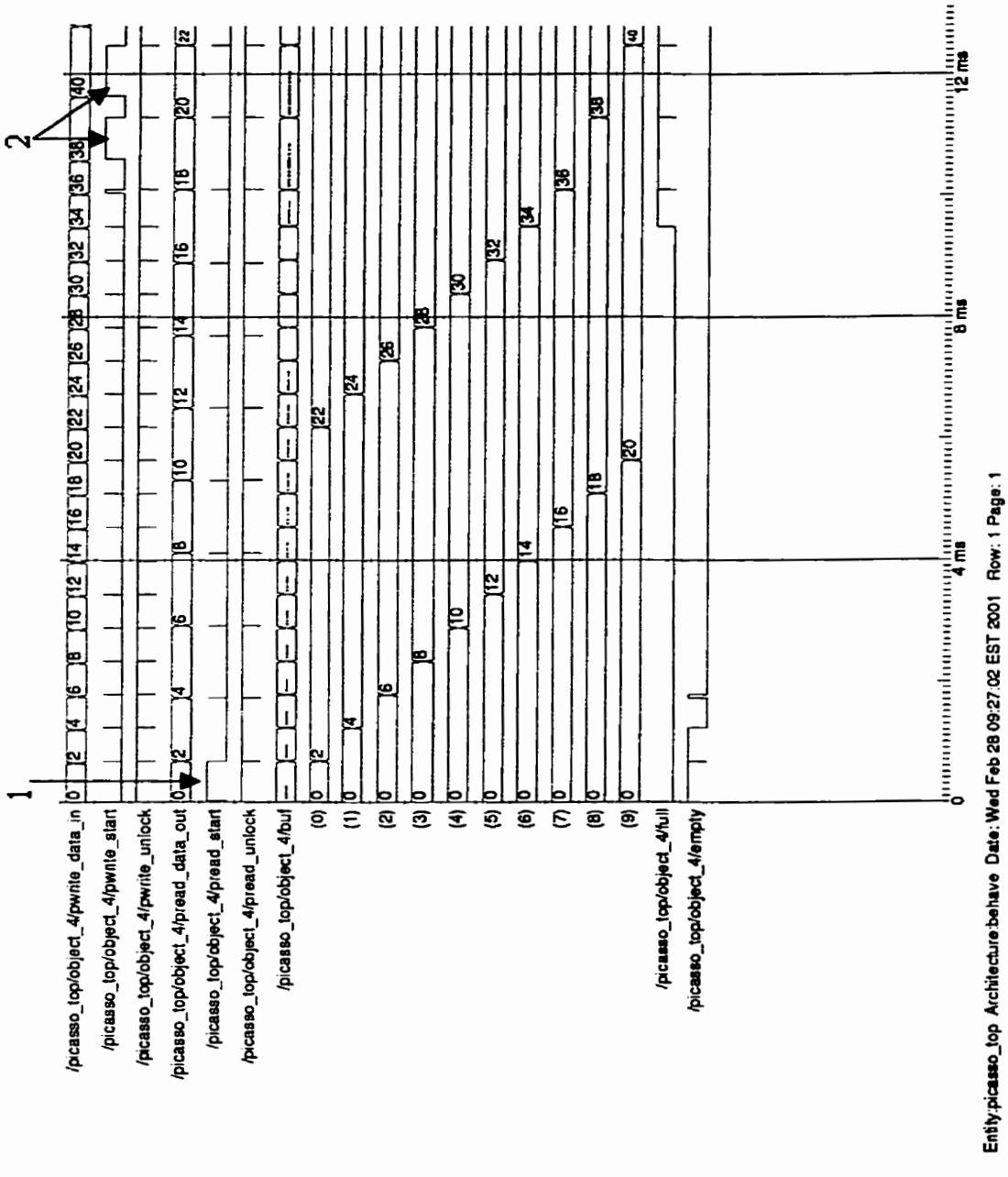

Figure 6.6 Traces de simulation du FIFO

Une fois la validation complétée avec Seamless, le système peut être synthétisé avec Leonardo ou un autre outil de synthèse RTL, comme on l'a vu à la section 2.7. Le système obtenu après synthèse pourra être co-simulé à nouveau avec Seamless.

Cet exemple permet de montrer le bon fonctionnement de Picasso et sa capacité à générer un système complet à partir d'une description haut niveau. Les extensions au langage permettent de modéliser la communication bloquante (RPC de Coware) mais également d'autres mécanismes. En effet, nous aurions pu bâtir un FIFO semblable sans blocage (la perte de données est donc possible si le FIFO est plein). Modéliser un système non bloquant avec des RPC uniquement aurait été plus complexe.

6.2 L'additionneur

Au chapitre trois, deux versions d'additionneur, l'une en matériel et l'autre en logiciel, furent présentées. Également, les deux clients servants de banc de test ont été vus. Grâce à ces composants, on peut bâtir le système de la Figure 6.7. Ce système peut être vu comme deux sous-systèmes concurrents. Dans un, le client logiciel demande au bloc matériel de faire l'addition de deux valeurs. Dans l'autre sous-système, c'est l'inverse qui se produit.

Le bloc « clocker » n'est utilisé que pour générer les signaux d'horloge et de remise à zéro pour les deux blocs matériels.

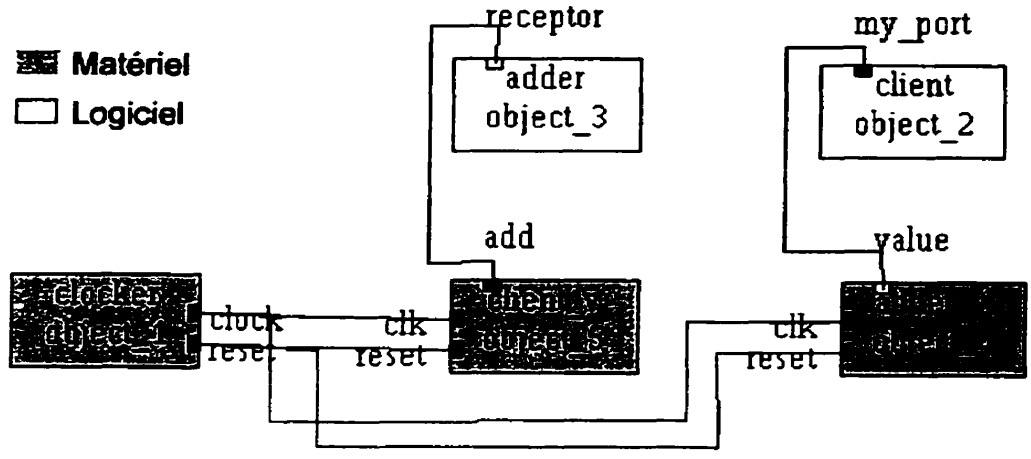

Figure 6.7 Système à deux additionneurs

Les résultats présentés au Tableau 6-1 illustrent le système généré par Picasso. L'intérêt ici est de comparer le nombre de lignes et le nombre de fichiers entrés par l'usager par rapport au système généré par Picasso avec un ou deux microprocesseurs. Il y a environ un facteur de 15 séparant le nombre de lignes de la spécification abstraite de l'usager et le nombre de lignes du système final. Picasso permet donc de réduire le nombre de lignes codées pour modéliser et éventuellement synthétiser un système.

Tableau 6-1 Taille de l'exemple avant et après génération

	Nombre de lignes	Nombre de fichiers
Spécification dans Picasso	112	6
Système avec 1 i960	1666	22
Système avec 2 i960	2026	27

6.3 Le réseau de communication

L'exemple suivant consiste à modéliser un petit réseau composé d'un routeur logiciel avec des clients et des serveurs qui peuvent être en logiciel ou en matériel. Un des objectifs est de montrer que l'environnement Picasso permet la réutilisation. On suppose que des composants matériels clients et serveurs sont déjà conçus pour communiquer entre elles selon un certain protocole. Nous voulons réutiliser ces composants en les rendant capables de communiquer en réseau, dans une approche co-design. Cela implique de concevoir des objets capables de lier le protocole du réseau au protocole utilisé par Picasso. C'est à ce moment que l'utilisation des descriptions mixtes devient utile. Il suffit de créer une seule fois un bloc qui accomplit cette tâche de conversion. On peut ensuite le réutiliser pour chaque client

La Figure 6.8 présente une vue d'ensemble du système. L'objet appelé « clocker » est un bloc matériel qui agit comme générateur d'horloge et de remise à zéro pour les différents blocs matériels utilisés. Le bloc routeur en logiciel reçoit les appels des clients et des serveurs sur ses différents ports. On sépare les ports de donnée et les ports de contrôle. Les ports « data » servent aux transferts des données entre les différents clients et serveurs. Les ports « calling » permettent d'appeler un serveur tandis que les ports « comm » reçoivent les appels de nouvelles connexions en provenance des clients.

Les objets « tel1 », « tel2 » et « tel3 » sont des objets hiérarchiques qui contiennent les clients ou les serveurs. Dans notre cas, « tel1 » est un client matériel, « tel2 » est un serveur en matériel et « tel3 » est un serveur en logiciel.

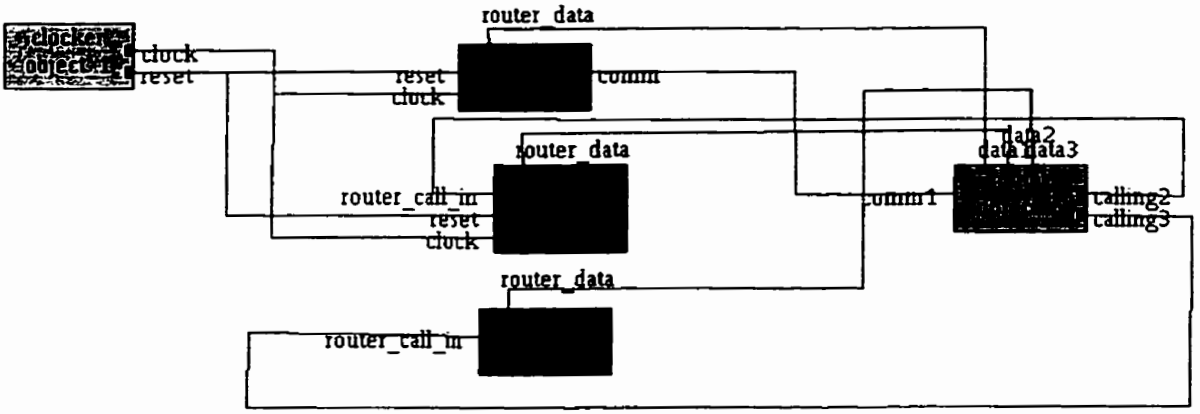

Figure 6.8 Le routeur

Si on regarde la structure de « tel1 » (Figure 6.9), on y trouve l'objet « client » et l'objet « inter » qui sont tous deux en matériel. « client » représente l'objet que nous réutilisons. Il est capable d'envoyer et de recevoir des données selon son protocole. Il ne dispose que de ports VHDL. « Inter » est l'interface qui permet de traduire le protocole du client vers un protocole de haut niveau, en utilisant des ports maîtres et esclaves. On voit donc que le même objet possède à la fois des ports VHDL et des ports de haut niveau. On peut donc réutiliser cet objet pour chaque client matériel.

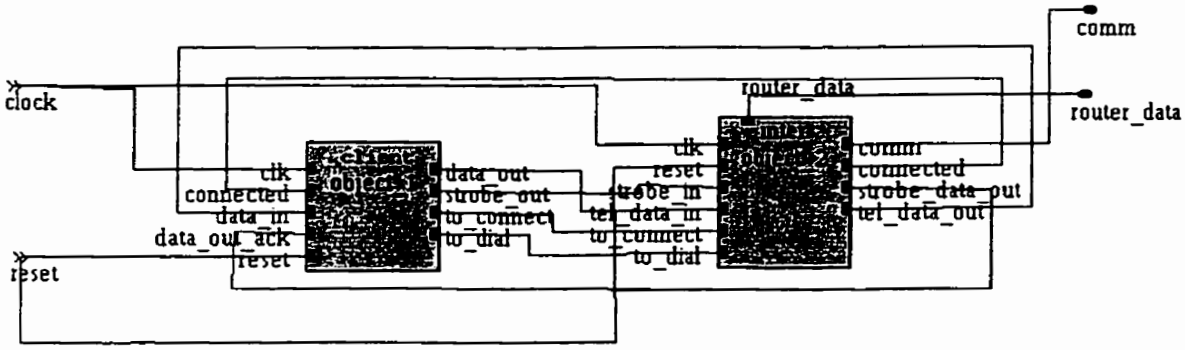

Figure 6.9 Structure de Tel1

Nous allons maintenant examiner le « client ». Les principaux signaux sont résumés au tableau suivant :

Tableau 6-2 Signaux du client

Nom	Type	Description
clk	in std_logic	Horloge à utiliser pour se synchroniser
Connected	in std_logic	Indique si on est connecté au serveur
data_in	in std_logic_vector(31 downto 0)	Bus pour les données reçues
data_out	out std_logic_vector(31 downto 0)	Bus pour les données envoyées
data_out_ack	in std_logic	Signal qu'une donnée est disponible pour être lue
reset	in std_logic	Remise à zéro du système
strobe_out	out std_logic	Signal qu'une donnée est prête à être envoyée
to_connect	out std_logic	Indique que l'on veut se connecter
to_dial	out std_logic_vector(15 downto 0)	Indique le numéro du serveur que l'on désire rejoindre

La procédure de connexion du client est la suivante : le client envoie tout d'abord le numéro du serveur auquel il veut se connecter sur son bus de données. Il signale cette information par la pair de signaux « to_connect » et « strobe_out » qui agissent comme protocole *request/acknowledge*. Le client attend ensuite que la connexion soit établie. L'entrée « connected » passera à '1' lorsque ce sera le cas.

Une fois connectés, le client et le serveur peuvent échanger des données. L'échange de données utilise encore un protocole de type requête/acquittement *(request/acknowledge)* avec les signaux « strobe_out » et « data_out_ack ». Dans notre cas, le protocole agit en échangeant un octet contre un octet. Ce choix est utile pour simplifier le code VHDL afin de garder l'attention sur les mécanismes de communication et non sur la fonctionnalité globale du système.

Enfin, lorsque le client ramène sa ligne « to_connect » à '0', la connexion est brisée et le serveur redevient disponible.

Le bloc « inter » sert d'interface. Il permet de passer les données d'un protocole à l'autre. Lorsque le client tente de se brancher sur un serveur, les valeurs lues sur le port de type VHDL sont renvoyées vers le routeur par une opération *Write*. Puis, l'attente de connexion se fait par le *Start*. Lorsque cette fonction débloque, la connexion est établie et le client est averti.

Le module *Inter* entre ensuite en régime permanent. Il transfert les données en utilisant le protocole haut niveau de Picasso composé des *read*, *write* et *start* et du protocole bas niveau avec les signaux VHDL. Les données reçues du client sont envoyées au serveur par la commande *write*. Le serveur est averti par la commande *start*. Quand celui-ci termine, la valeur de retour du serveur, lu par la commande *read*, est renvoyée au client.

On pourrait procéder à une analyse similaire pour le deuxième module hiérarchique, appelé « tel2 ». Ce module représente un serveur en matériel qui augmente de un la valeur reçue. Le module d'interface « inter_slave » permet de brancher un serveur au routeur. Quant au bloc « tel3 », celui-ci contient un serveur en logiciel qui procède à des multiplications par deux. On est donc dispensé de créer une interface matérielle pour ce bloc. Picasso en créera une si ce bloc ne tourne pas sur le même microprocesseur que le routeur.

Le dernier composant est le routeur. Pour chaque client branché, deux ports esclaves sont réservés. Le premier est appelé « *comm* » et reçoit les informations pour établir une nouvelle connexion. Le deuxième port esclave, « *data* », reçoit les données du client et les renvoie vers le serveur. Deux ports maîtres sont réservés pour chaque serveur. Le premier s'appelle « *calling* » et permet d'appeler un serveur. Le deuxième, « *data* », sert à envoyer les données du client et recevoir les données du serveur. La tâche du routeur est donc de brancher deux ports « *data* » ensembles pour que la communication soit possible.

On peut générer ce système pour un microprocesseur. Le Tableau 6-3 résume le nombre de lignes et le nombre de fichiers des spécifications entrées par l'usager, comparés à ceux du système généré par Picasso.

Tableau 6-3 Comparaison du routeur avant et après génération

	Nombre de lignes	Nombre de fichiers
Spécification dans Picasso	651	9
Système avec un i960	2286	24

6.4 Analyse des résultats

À la lumière des résultats précédents, on constate que Picasso vient aider à la conception en diminuant d'un facteur de 10 à 15 LOC (de l'anglais, *ligne of code*) la spécification d'un système. Cela vient du fait que le niveau d'abstraction offert par Picasso est supérieur. L'avantage est d'avoir une spécification portable. Le risque d'erreur introduit par l'usager dans la spécification est réduit puisqu'il y a moins de lignes de code.

L'exploration de différente architecture est simplifiée. Si on dispose des représentations en logiciel et en matériel pour chaque module, on peut bâtir facilement un ensemble de systèmes complets et les départager les unes contre les autres en les simulant.

D'autres architectures de communication seront nécessaires pour bien couvrir les différentes possibilités d'implantation. Ainsi l'utilisation d'interruptions pour remplacer la phase où l'on scrute (*polling*) permettrait un gain en temps d'exécution. Une autre amélioration est d'utiliser une mémoire pour remplacer les registres. Cela permettrait de diminuer la superficie requise lorsqu'on a affaire à de grands ensembles de données.

Chapitre 7 - Conclusion

À titre de conclusion, ce chapitre met en relief la réalisation de nos principaux objectifs fixés dans ce travail. Quelques idées de travaux futurs sont également proposées, reprenant différentes suggestions faites tout au long du mémoire.

Au chapitre 1, nous avons vu d'abord ce qu'étaient les systèmes embarqués et les outils de co-design. Cela nous a permis de fixer des objectifs et critères pour réaliser une méthodologie cherchant à simplifier ce problème. On a vu au chapitre 1 que pour bien décrire un système dans une approche co-design, il faut utiliser des représentations offrant des niveaux d'abstraction élevés et offrir une méthodologie la plus complète qui soit.

Ces objectifs ont été raffinés au chapitre 2. Nous y avons présenté une revue de littérature visant à situer notre méthodologie dans le monde du co-design, pour permettre de préciser les objectifs principaux. Cette revue de littérature nous a permis d'établir un certain nombre d'objectifs à atteindre : 1) Générer automatiquement des architectures de système à partir d'une description où le partitionnement est connu. 2) Concevoir des mécanismes de communication de haut niveau, indépendants de la nature des modules. L'utilité de cet objectif est de brancher du logiciel avec du matériel ou d'autres blocs logiciels. La communication devient indépendante de la nature des modules. 3) Créer des interfaces pour établir des liens de communication avec des processeurs. Cela est requis pour permettre d'encapsuler les processeurs en bibliothèque et rendre leur utilisation plus facile par la méthodologie. 4) Pouvoir explorer facilement différentes architectures, nous avons vu comment en changeant les paramètres de construction, on peut créer des architectures différentes et les simuler

Au chapitre 3, on a vu comment les spécifications sont définies grâce à notre méthodologie. On a montré comment nous décrivions un système à haut niveau en

utilisant une syntaxe capable d'abstraire les communications pour rencontrer l'objectif #2. Cette syntaxe faisant partie également de modifications aux langages C et VHDL. On a également montré comment la saisie de spécification pouvait être indépendante de la nature des modules en utilisant la notion de port et d'interface. Chaque langage est capable d'exprimer la notion de port, ce qui permet de séparer la communication du comportement.

Au chapitre 4, nous avons montré un exemple d'architecture conçue en vue d'être facilement modifiable et réutilisable. Cela a permit de définir la notion de Ichip et archi permettant de bâtir la bibliothèque de processeurs et d'atteindre l'objectif #3. Ces modules permettent à la méthodologie d'implanter la communication de haut niveau sur un système réel.

Au chapitre 5, nous avons montré comment les spécifications de haut niveau pouvaient être traduites pour permettre la construction de système. Cela a permit de rencontrer l'objectif #1 et de montrer comment on peut générer des systèmes automatiquement et comment passer d'une architecture a l'autre pour l'exploration de système en modifiant les paramètres de construction. Ainsi, ajouter un deuxième processeur vient modifier considérablement l'architecture finale mais son ajout à haut niveau est très facile.

Le chapitre 6 a permis de renforcer ces justifications en présentant des exemples d'utilisations typiques où des systèmes ont été capturés, générés et vérifiés avec le logiciel Seamless. Cela a permis de vérifier la réalisation de l'objectif #4.

7.1 Travaux futurs

Quelques travaux futurs ont été soulignés tout au long de mémoire. On peut maintenant en dresser une liste synthèse visant à guider le futur développement de la méthodologie.

Premièrement, le concept de partitionnement devrait être automatique et utiliser la notion d'estimateurs. Pour réaliser cet objectif, un seul langage devrait être utilisé pour décrire aussi bien du logiciel et du matériel. Ce même langage devrait également supporter la notion de contrainte pour permettre à des estimateurs de vérifier quelle est la meilleure implantation pour une spécification donnée. On pourrait ainsi exprimer la notion de « contrainte de précédance » pour relier les différents blocs fonctionnels aux contraintes de temps. Cela est en soi un domaine de recherche très complexe.

En deuxième lieu, il serait intéressant d'avoir des mécanismes de communication plus complexes à haut niveau et une représentation de ceux-ci à bas niveau. Par exemple, des mécanismes pourraient permettre de définir des messages, de faire des appels bloquant ou non. Cela permettrait d'obtenir des spécifications beaucoup plus près du langage et de simplifier le travail de conception.

On devrait également penser à utiliser un système d'exploitation temps réel standard. Cela permettrait d'avoir beaucoup plus de raffinement et éviterait d'avoir à concevoir nous-mêmes notre propre système. Cela diminuerait la courbe d'apprentissage des concepteurs en utilisant des composants déjà connus.

Enfin, il faudrait développer la notion de Ichip et archi pour d'autres mécanismes de communication et d'autres processeurs.

Bibliographie

[1] "i960 KA/KB Microprocessor Programmer's Reference Manual", Intel Corporation, 1991

[2] "System Level Design Language", Available at http://www.inmet.com/sldl/

[3] "SystemC user guide version 1.1", www.systemc.org

[4] "SystemC" v1.1, User Guide, Available on http://www.systemc.org

[5] Adams, J.K., Thomas DE.., "The Design of Mixed Hardware/Software Systems". Design automation conference, 1996, pp.515 – 519

[6] Allen R., Gajski D., "The Case for C/C++ Hardware Design", EEDesign, June 16, 2000.

[7] Arnout G., "System C Standard", Proc. Of Asia-Pacific DAC, January 2000, Also available on http://www.coware.com/pdf/aspdac.pdf

[8] Arnout, G., "SystemC standard", Asia-Pacific DAC, janvier 2000

[9] Dataquest Inc., http://www.mentor.com/press_releases/jan00/seamless_pr.html

[10] Doucet F., Vivek S., Gupta R., "system-on-Chip Modeling Using Objects and Their Relationships", http://www.ics.uci.edu/~iesag/

[11] Elliott J. P., Understanding Behavioral Synthesis A practical Guide to High Level Synthesis, Kluwer Academic Publishers, 1999.

[12] Gajski D., Vahid F., Narayan S., "Specification and Design of Embedded Systems", Prentice Hall, 1994, chapter 3.

[13] Henkel J., Benner Th, Ernst R., "Hardware generation and partitioning effects in the COSYMA system.", Handouts from IEEE/ACM International Workshop on hardware-software codesign, Cambridge, Massachusetts, 7-8 octobre 1993

[14] Herrmann D., Henkel J., "An approach to the adaptation of estimated cost parameters in the cosyma system", www.ida.ing.tu-bs.de

[15] http://www.faqs.org/rfcs/rfc1014.html

[16] http://www.ireste.fr/mcse/htmlan/example-0.html

[17] http://www.mentorg.com/renoir/datasheets/rends.pdf

[18] http://www.mentorg.com/seamless/datasheet/seamless_ds.pdf

[19] Jerraya, A.A., Romdhani M., "Multilanguage specification for system design and codesign", demander a Guy pour référence

[20] Kamal Hashmi M.M., "ICL: VHDL+ Language Reference Manual, Available on http://www.icl.com/da

[21] Ku D.,De Micheli G., "HardwareC: A language for hardware design", Computer system laboratory, Stanford University, Aout 1990

[22] Lavagno L., Sangiovani-Vincentelli A. "Embedded system codesign: synthesis and verification", Kluwer Academic, Boston, MA, 1996, page 213-242

[23] Lennard C.K., Schaumont P., Gong J., "Standards for System-Level Design: Pratical Reality or Solution in Search of a Question?", Proc. of DATA 2000, Also available on http://www.coware.com/pdf/DATE.pdf.

[24] Lennard C.K., System-Level Interface Behavioral Documentation Standard, VSI Alliance™ , 1998

[25] Lennard, C.K., "Standards for system-level design: practical reality or solution in search of a question", www.vsi.org

[26] Li, Y.-T., Malik, S., "Performance Analysis of Real-Time Embedded Software", Kluwer Academic Publishers, 1999.

[27] Lin B., Rompaey K.V., "Designing Single Chip Systems", www.coware.com

[28] Lipman J., "Chip hardware and software" EDN magazine, july 1996, pp.75

[29] Narayan S., Gajski D., "Features supporting system-level specification in HDLs", EuroDac, 1993

[30] Niemann R., "Hardware/Software Co-Design for Data Flow Dominated Embedded Systems" Kluwer Academic Publishers, 1998.

[31] Page I., "Closing the gap between hardware and software: hardware-software cosythesis at Oxford" IEEE Colloqium on Hardware-Software Cosynthesis for reconfigurable Systems. Bristol, UK, fevrier 1996.

[32] Rompaey K.V., Verkest D., "CoWare- A design for heterogeneous hardware/software systems", www.coware.com

[33] Roth R.,Ramanathan D., "A High-level hardware design methodology using C++", www.cynapps.com.

[34] Staunstrap J., Wolf W., "Hardware/Software Co-design: Principles and Practice", Kluwer Academic Publishers, 1997, pp.113-148

[35] Steven E. S., "The New System-Level Design Language, Integrated System Design, July 1998

[36] Weil, P. "EDA Roadmap Task Force Report : Design of Microprocessors" Electronic Design Automation Industry Council, 1999

[37] Zorian Y., Marinissen E., "Testing embedded-core-based system chips", Computer magazine from IEEE, june 1999, p.52-54

Annexe 1 - i_chip.h

```
#include <setjmp.h>

#include <stdlib.h>
#define     Pause       if
(!setjmp(thread_env[GlobalThread])){change_thread();}
extern int GlobalThread;
extern jmp_buf thread_env[nb_thread];
void change_thread();
void send(int pCopro);
void io_polling();
void Lock(int i);
void Unlock(int i);
void init_thread();
void StartEngine();
typedef void (*Tthread_func)();
void init();

#define polling_mem *(int*)0x60000000
#define ctrl_mem *(int*)0x50000000

#define true 1
#define false 0
```

Annexe 2 - data_type.h

```
typedef long PIC_int;

typedef char PIC_bool;

typedef struct _PIC_sum{
      PIC_int     x;
      PIC_int     y;
} PIC_sum;
```

Annexe 3 - i960_1_code.c

```
/* generated by picasso, 1999, 2000 */
#include "data_type.h"
#include "i_chip.h"


#define object_2_my_port_data_in (*(PIC_int*)1073741824)

#define object_2_my_port_data_out (*(PIC_sum*)1073741828)

#define object_3_receptor_data_in (*(PIC_sum*)1073741836)

#define object_3_receptor_data_out (*(PIC_int*)1073741844)



void my_thread_thread()
{
      for(;;){
    PIC_int result;

    object_2_my_port_data_out.x=40;
    object_2_my_port_data_out.y=50;
send(1);
Pause;

    result=(object_2_my_port_data_in);

    Pause;
}
}



void object_3_receptor_slave()
{
      for(;;){

object_3_receptor_data_out=(object_3_receptor_data_in.x+(object_3_recep
tor_data_in.y));
send(2);
Pause;
}
}



void init() {}
```

```
Tthread_func
thread_func[]={NULL,object_3_receptor_slave,my_thread_thread};

void main()
{
      init_thread();

      /*lock the slave thread*/
      Lock(1);

      StartEngine();
}
```

Annexe 4 - i_chip.c

```
#include "i_chip.h"

jmp_buf thread_env[nb_thread];
int thread_started[nb_thread]={0};
int thread_lock[nb_thread]={0};
int GlobalThread;
char thread_stack[nb_thread][1024*thread_stack_size];
char *thread_stack_ptr[nb_thread];
int ID_comm[nb_thread];

extern Tthread_func thread_func[];

void change_thread()
{
      do {
            GlobalThread=GlobalThread+1;
            if (GlobalThread==nb_thread){
                  io_polling();
                GlobalThread=0;
            }
      } while (thread_lock[GlobalThread]==1 ||
!thread_func[GlobalThread]);

      if (!thread_started[GlobalThread]){
        asm(int,"ld _GlobalThread,g0","ld _thread_stack(g0),fp","addi
24,fp,sp");
        thread_started[GlobalThread]=1;
        thread_func[GlobalThread]();
      } else {
        longjmp(thread_env[GlobalThread],1);
      }
}

void send(int pCopro)
{
      ctrl_mem=pCopro;
      Lock(GlobalThread);
        ID_comm[pCopro-1]=GlobalThread;
}

void Lock(int i)
{
      thread_lock[i]=1;
}

void Unlock(int i)
{
      thread_lock[i]=0;
}

void io_polling()
```

```
{
      for(;;){
            int value=polling_mem;
            if (!value){
                  break;
            } else {
                   Unlock(ID_comm[value-1]);
            }
            polling_mem=value;
      }
}

void init_thread()
{
        /*init the thread*/
        int i;
      for (i=0;i<nb_thread;i++){
            if (thread_func[i]){
                  thread_stack_ptr[i]=thread_stack[i]+64-
((long)thread_stack[i])%64;
            }
                ID_comm[i]=i;
      }
}

void StartEngine()
{
      init();
      /*let start!!*/
      GlobalThread=-1;
      change_thread();
}
```

Annexe 5 - copro_type_i960_1_code.vhd

```
-- generated by picasso, 1999, 2000

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_signed.all;

use work.data_type.all;

use work.arch_type_i960_1_code.all;


entity copro_type_i960_1_code is

port(

      object_2_my_port_data_in : in PIC_int:=0;

      object_2_my_port_data_out : inout PIC_sum:=(0,0);

      object_3_receptor_data_in : in PIC_sum:=(0,0);

      object_3_receptor_data_out : inout PIC_int:=0;

      copro_mem_cs     : in std_logic;

      address          : in address_bus_i960_1_code;

      copro_data_in    : in data_bus_i960_1_code;

      copro_data_out   : out data_bus_i960_1_code;

      w_rbar           : in std_logic

      );

end copro_type_i960_1_code;


architecture behave of copro_type_i960_1_code is

begin

      process(copro_mem_cs,address,copro_data_in,w_rbar,object_2_my_port
_data_in,object_2_my_port_data_out,object_3_receptor_data_in,object_3_r
```

```
eceptor_data_out)

      begin

            copro_data_out<=(others=>'0');

            if copro_mem_cs='1' then

                  if (w_rbar='1') then

                        case
conv_integer(address(address_bus_width_i960_1_code-9 downto 0)) is

                        when 1=>
object_2_my_port_data_out.x<=conv_integer(copro_data_in);

                        when 2=>
object_2_my_port_data_out.y<=conv_integer(copro_data_in);

                        when 5=>
object_3_receptor_data_out<=conv_integer(copro_data_in);

                        when others => NULL;

                        end case;

                  else

                        case
conv_integer(address(address_bus_width_i960_1_code-9 downto 0)) is

                        when 0=>
copro_data_out<=conv_std_logic_vector(object_2_my_port_data_in,data_bus
_width_i960_1_code);

                        when 1=>
copro_data_out<=conv_std_logic_vector(object_2_my_port_data_out.x,data_
bus_width_i960_1_code);

                        when 2=>
copro_data_out<=conv_std_logic_vector(object_2_my_port_data_out.y,data_
bus_width_i960_1_code);

                        when 3=>
copro_data_out<=conv_std_logic_vector(object_3_receptor_data_in.x,data_
bus_width_i960_1_code);

                        when 4=>
copro_data_out<=conv_std_logic_vector(object_3_receptor_data_in.y,data_
bus_width_i960_1_code);

                        when 5=>
copro_data_out<=conv_std_logic_vector(object_3_receptor_data_out,data_b
```

```
us_width_i960_1_code);
                        when others => NULL;
                        end case;
                    end if;
                end if;
        end process;
end behave;
```

Annexe 6 - address_decode.vhd

```
--------------------------------------------------------------------------------
--
--    address_decode.vhd
--
--    This is the address_decode for the memory controller of i960KX
--    testbench.
--
--    Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                        UNPUBLISHED, LICENSED SOFTWARE.
--           CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--           PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
--------------------------------------------------------------------------------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;

entity address_decode is
    port(
         addr_in          : in   std_logic_vector( 31 downto 4 );
         bank_0_cs        : out  std_logic := '1';
         bank_1_cs        : out  std_logic := '1';
       picasso_io_cs    : out  std_logic := '1';
       cs_fault     : out  std_logic := '1'
        );

end address_decode;

architecture behave of address_decode is
begin
    process( addr_in )
    begin

        -- This design is modeled with 1/2 Mbyte static RAMs (18 bit
address bus)
        -- originating at address 0. Set the chip select for these
static
        -- RAMs based on address bus in.

        if (addr_in(31 downto 21) = "00000000000") then
          picasso_io_cs <= '1';
          cs_fault <= '1';
          if (addr_in(20) = '0') then
              bank_0_cs <= '0';
              bank_1_cs <= '1';
          elsif (addr_in(20) = '1') then
              bank_0_cs <= '1';
              bank_1_cs <= '0';
```

```
        end if;
      elsif (addr_in(31 downto 30) = "01") then
        picasso_io_cs <= '0';
          bank_0_cs <= '1';
          bank_1_cs <= '1';
        cs_fault <= '1';
      else
        picasso_io_cs <= '1';
          bank_0_cs <= '1';
          bank_1_cs <= '1';
        cs_fault <= '0';
      end if;

    end process;
end behave;
```

Annexe 7 - address_latch.vhd

```
--------------------------------------------------------------------
--------
--
--    address_latch.vhd
--
--    This is the address_latch for the memory controller of i960KX
--    testbench. This latches the address bus on the falling edge of
ALE.
--    The four least significant bits are not latched because they are
--    controlled by the burst logic.
--
--    Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                        UNPUBLISHED, LICENSED SOFTWARE.
--          CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--          PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
--------------------------------------------------------------------
--------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;

entity address_latch is
    port(
         ale_bar          : in   std_logic;
         addr_in          : in   std_logic_vector( 31 downto 4 );
         addr_out         : out  std_logic_vector( 31 downto 4 ) :=
                              "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
        );

end address_latch;

architecture behave of address_latch is
begin
    process( ale_bar )
    begin
        if (ale_bar = '0') then
            addr_out <= addr_in;
        end if;
    end process;
end behave;
```

Annexe 8 - burst_logic.vhd

```
--------------------------------------------------------------------------------
--
--    burst_logic.vhd - Control burst data cycles for i960kx.
--
--    This is the address_latch for the memory controller of i960KX
--    testbench.
--
--    Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                        UNPUBLISHED, LICENSED SOFTWARE.
--          CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--          PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
--------------------------------------------------------------------------------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use ieee.std_logic_1164_extensions.all;

entity burst_logic is
    port(
         addr_in           : in   std_logic_vector( 3 downto 0 );
         den_bar           : in   std_logic;
         ads_bar           : in   std_logic;
         ready_bar         : in   std_logic;
         cycle_in_progress : out  std_logic := '0';   -- asserted high
         addr_3_2          : out  std_logic_vector( 3 downto 2 ) :=
"00"
        );

end burst_logic;

architecture behave of burst_logic is

begin
    process

      variable size_bits : std_logic_vector( 1 downto 0 );
      variable addr_bits : std_logic_vector( 3 downto 2 );

    begin

        -- If ADS is asserted (low) and DEN is not asserted (high),
        -- then we are at the beginning of a bus cycle. We initialize
our
        -- counters and assert (high) cycle_in_progress.
        if ((ads_bar /= '0') or (den_bar /= '1')) then
            wait until ((ads_bar = '0') and (den_bar = '1'));
```

```
        end if;

        size_bits := addr_in(1 downto 0);    -- initialize from LAD[1:0]
        addr_bits := addr_in(3 downto 2);    -- initialize from LAD[3:2]

        addr_3_2 <= addr_bits;
        cycle_in_progress <= '1';

        wait until (ready_bar'event and (ready_bar = '1'));

        while (size_bits /= "00") loop
            size_bits := unsigned (size_bits) - 1;
            addr_bits := unsigned (addr_bits) + 1;
            addr_3_2 <= addr_bits;
            wait until (ready_bar'event and (ready_bar = '1'));
        end loop;

        cycle_in_progress <= '0';    -- deassert

        wait on ads_bar, den_bar;
    end process;
end behave;
```

Annexe 9 - byte_enable_latch.vhd

```
--------------------------------------------------------------------------------
--
--   byte_enable_latch.vhd
--
--   This is the byte enable_latch for the memory controller of i960KX
--   testbench.
--
--   Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                        UNPUBLISHED, LICENSED SOFTWARE.
--          CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--          PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
--------------------------------------------------------------------------------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;

entity byte_enable_latch is
    port(
         ale_bar          : in   std_logic;  -- address latch enable
from cpu
         ready_bar        : in   std_logic;  -- ready from mem
controller
         be_in_bar        : in   std_logic_vector( 3 downto 0 ); -- in
from cpu
         be_out_bar       : out  std_logic_vector( 3 downto 0 ) :="1111"
-- out to mem
        );

end byte_enable_latch;

architecture behave of byte_enable_latch is

begin

    process

    variable next_be : std_logic_vector(3 downto 0) := "1111";

    begin


        if (ale_bar'event and (ale_bar = '0')) then
            next_be := be_in_bar;
            be_out_bar <= be_in_bar;
        end if;

        if (ready_bar'event) then
```

```
            if (ready_bar = '1') then
                be_out_bar <= next_be;
            else
                next_be := be_in_bar;
            end if;
        end if;

    wait on ale_bar, ready_bar;

    end process;

end behave;
```

Annexe 10 -sram_if.vhd

```
------------------------------------------------------------------
--------------------
--
--   sram_if.vhd - output control lines to SRAM for i960KX mem
controller.
--
--   This is the address_latch for the memory controller of i960KX
--   testbench.
--
--   Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                      UNPUBLISHED, LICENSED SOFTWARE.
--         CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
------------------------------------------------------------------
--------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;

entity sram_if is
    port(
         oe_in_bar          : in   std_logic;
         we_in_bar          : in   std_logic;
         be_bar             : in   std_logic_vector( 3 downto 0 );
         oe_out_bar         : out  std_logic_vector( 3 downto 0 ):=
"1111";
         we_out_bar         : out  std_logic_vector( 3 downto 0 ):=
"1111"
        );

end sram_if;

architecture behave of sram_if is

begin
    process( be_bar, oe_in_bar, we_in_bar )
    begin

        oe_out_bar(3) <= oe_in_bar or be_bar(3);
        oe_out_bar(2) <= oe_in_bar or be_bar(2);
        oe_out_bar(1) <= oe_in_bar or be_bar(1);
        oe_out_bar(0) <= oe_in_bar or be_bar(0);

        we_out_bar(3) <= we_in_bar or be_bar(3);
        we_out_bar(2) <= we_in_bar or be_bar(2);
        we_out_bar(1) <= we_in_bar or be_bar(1);
        we_out_bar(0) <= we_in_bar or be_bar(0);
```

```
    end process;
end behave;
```

Annexe 11 -timing_control.vhd

```
--------------------------------------------------------------------------------
--
--   timing_control.vhd - Control sram timing and READY generation for
--     i960kx.
--
--   This is the address_latch for the memory controller of i960KX
--   testbench.
--
--   Copyright (c) MENTOR GRAPHICS CORPORATION 1995 All Rights Reserved
--                        UNPUBLISHED, LICENSED SOFTWARE.
--         CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
--         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
--
--------------------------------------------------------------------------------

library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use ieee.std_logic_1164_extensions.all;

entity timing_control is
    port(
         sram_cs_bar        : in   std_logic;
         write_read_bar     : in   std_logic;
         clk2               : in   std_logic;
         den_bar            : in   std_logic;
         cycle_in_progress  : in   std_logic;
         ready_bar          : out  std_logic := '1';
         sram_oe_bar        : out  std_logic := '1';
         sram_we_bar        : out  std_logic := '1'
        );

end timing_control;

architecture behave of timing_control is

begin
    process
    begin

        if ((cycle_in_progress = '1') and (sram_cs_bar = '0')) then

            wait until (clk2'event and (clk2 = '1')); -- 2nd hi clk2 of
Ta
            wait until (den_bar = '0');
            case write_read_bar is
                when '1' =>
                    sram_oe_bar <= '1';
```

```
                    sram_we_bar <= '0';
                when '0' =>
                    sram_we_bar <= '1';
                    sram_oe_bar <= '0';
                when others =>
                    sram_oe_bar <= '1';
                    sram_we_bar <= '1';
            end case;

            wait until (clk2'event and (clk2 = '1')); -- 2nd hi clk2 of
Td

            while (cycle_in_progress = '1') loop

                -- insert wait state processing here

                if (write_read_bar = '1') then
                    sram_we_bar <= '0';
                    wait for 1 ns;
                end if;
                ready_bar <= '0';                              -- Assert
READY
                wait for 0 ns;      -- force delta to propagate value
                wait until (clk2'event and (clk2 = '1')); -- 1st hi
clk2 Td

                -- The Miami srams latch the data on writes on the
deassertion
                -- of either CE_ or WE_. This next if statement makes
sure
                -- that WE_ is deasserted before LAD is released.

                if (write_read_bar = '1') then
                    sram_we_bar <= '1';
                end if;

                wait until (clk2'event and (clk2 = '0')); -- 1st lo
clk2 Td
                ready_bar <= '1';                              -- deassert
READY

                wait until (clk2'event and (clk2 = '1')); -- 2nd hi
clk2 Td

            end loop;

            sram_oe_bar <= '1';
            sram_we_bar <= '1';

        end if;

        wait on cycle_in_progress, sram_cs_bar;
      wait for 1 ns;
    end process;
```

```
end behave;
```

Annexe 12 -arch_type_i960_1_code.vhd

```
library ieee;

use ieee.std_logic_1164.all;


package arch_type_i960_1_code is

constant nb_copro_i960_1_code:integer:=2;

constant data_bus_width_i960_1_code:integer:=32;

constant address_bus_width_i960_1_code:integer:=28;


subtype data_bus_i960_1_code is
std_logic_vector(data_bus_width_i960_1_code-1 downto 0);

subtype address_bus_i960_1_code is
std_logic_vector(address_bus_width_i960_1_code-1 downto 0);

subtype copro_type_i960_1_code is integer range 0 to
nb_copro_i960_1_code+1;

type fifo_array_type_i960_1_code is array(copro_type_i960_1_code) of
copro_type_i960_1_code;

type copro_ask_type_i960_1_code is array(copro_type_i960_1_code) of
std_logic;

end arch_type_i960_1_code;
```

Annexe 13 -copro_master.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity copro_master is
    port(
         unlock          : out std_logic;
         clk             : in std_logic;
         reset           : in std_logic;
       copro_start       : in std_logic;
       copro_ask   : out std_logic;
       copro_ack   : in std_logic:='0';
       resume_copro      : in std_logic:='0'
        );

end copro_master;

architecture behave of copro_master is
begin
    master : process
    begin
    reset_loop:loop
        unlock<='1';
      copro_ask<='0';
      wait until clk'event and clk='0';
      if (reset='1') then exit reset_loop;end if;
      while (copro_start='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
      copro_ask<='1';
        while (copro_ack='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
      copro_ask<='0';
      while (resume_copro='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
        unlock<='0';
      while (copro_start='1') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;

    end loop reset_loop;
    end process master;
end behave;
```

Annexe 14 -copro_slave.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity copro_slave is
    port(
         unlock          : in std_logic;
         clk             : in std_logic;
         reset           : in std_logic;
       copro_start       : out std_logic;
       copro_ask   : out std_logic;
       copro_ack   : in std_logic:='0';
       resume_copro      : in std_logic:='0'
        );

end copro_slave;

architecture behave of copro_slave is
begin
    slave : process
    begin
    reset_loop:loop
      copro_start<='0';
      copro_ask<='0';
      wait until clk'event and clk='0';
      if (reset='1') then exit reset_loop;end if;
      while (resume_copro='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
      while (unlock='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
      copro_start<='1';
      while (unlock='1') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
      copro_start<='0';
      copro_ask<='1';
      while (copro_ack='0') loop
            wait until clk'event and clk='0';
            if (reset='1') then exit reset_loop;end if;
        end loop;
    end loop reset_loop;
    end process slave;
end behave;
```

Annexe 15 -architec_i960_1_code.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use work.arch_type_i960_1_code.all;

entity architec_i960_1_code is
    port(
         cs_bar           : in std_logic;
       oe          : in std_logic;
       we          : in std_logic;
         data             : inout  data_bus_i960_1_code:=(others=>'Z');
         copro_data_in    : out  data_bus_i960_1_code;
         copro_data_out   : in  data_bus_i960_1_code;
         address          : in  address_bus_i960_1_code;
         w_rbar           : in std_logic;
         clk              : in std_logic;
         reset            : in std_logic;

       copro_ask   : in copro_ask_type_i960_1_code;
       copro_ack   : out copro_ask_type_i960_1_code:=(others=>'0');
       resume_copro      : out
copro_ask_type_i960_1_code:=(others=>'0');
       copro_mem_cs      : out std_logic

        );

end architec_i960_1_code;

architecture behave of architec_i960_1_code is

signal fifo_select:std_logic:='0';
signal ctrl_select : std_logic:='0';
signal fifo_data_in   : data_bus_i960_1_code;
signal fifo_data_out  : copro_type_i960_1_code;

begin
    decode :
process(cs_bar,address,we,oe,data,fifo_data_out,copro_data_out)
    begin
      fifo_select<='0';
      ctrl_select<='0';
      copro_mem_cs<='0';
      fifo_data_in<=(others=>'0');
      data<=(others=>'Z');
      if cs_bar='0' and (we='0' or oe='0') then
           if address(address_bus_width_i960_1_code-1 downto
address_bus_width_i960_1_code-
2)="10" then
                fifo_select<='1';
```

```
                  if w_rbar='0' then

data<=CONV_STD_LOGIC_VECTOR(0,data_bus_width_i960_1_code-7) &
CONV_STD_LOGIC_VECTOR (fifo_data_out,7);
                  else
                        fifo_data_in<=data;
                  end if;
            elsif address(address_bus_width_i960_1_code-1 downto
address_bus_width_i960_1_code-
2)="01" then
                  ctrl_select<='1';
            else
                  copro_mem_cs<='1';
                  if w_rbar='0' then
                        data<=copro_data_out;
                  else
                        copro_data_in<=data;
                  end if;

            end if;
      end if;
    end process decode;

    fifo : process
      variable fifo_array:fifo_array_type_i960_1_code;
      variable index_fifo:copro_type_i960_1_code;
      variable index_copro:copro_type_i960_1_code;
    begin
      reset_loop : loop
            fifo_array:=(others=>0);
            index_fifo:=1;
            index_copro:=1;
            copro_ack<=(others=>'0');
            fifo_data_out<=0;
            loop
            wait until clk'event and clk='1';
            if (reset='1') then exit reset_loop;end if;
            fifo_data_out<=0;
            copro_ack<=(others=>'0');
            if (fifo_select='1') then
                  if (w_rbar='1') then
                        if  fifo_data_in=fifo_array(1) and index_fifo/=1
then
                              for i in 2 to fifo_array'length-1 loop
                                    fifo_array(i-1):=fifo_array(i);
                              end loop;
                              fifo_array(fifo_array'length-1):=0;
                              index_fifo:=index_fifo-1;
                        end if;
                  else
                        fifo_data_out<=fifo_array(1);
                  end if;
            else
                  if (copro_ask(index_copro)='1') then
```

```
                        fifo_array(index_fifo):=index_copro;
                        copro_ack(index_copro)<='1';
                        index_fifo:=index_fifo+1;
                  end if;
                  if (index_copro=fifo_array'length-1) then
                        index_copro:=1;
                  else
                        index_copro:=index_copro+1;
                  end if;
            end if;
            end loop;

      end loop reset_loop;
    end process fifo;

    ctrl_unit : process
    begin
      reset_loop : loop
      resume_copro<=(others=>'0');
      wait until clk'event and clk='1';
        if (reset='1') then exit reset_loop; end if;
      while (ctrl_select='0') loop
            wait until clk'event and clk='1';
            if (reset='1') then exit reset_loop; end if;
      end loop;
      resume_copro(conv_integer(data(7 downto 0)))<='1';
      wait until clk'event and clk='1';
        if (reset='1') then exit reset_loop; end if;
      resume_copro<=(others=>'0');
      while (ctrl_select/='0') loop
            wait until clk'event and clk='1';
            if (reset='1') then exit reset_loop; end if;
      end loop;
      end loop reset_loop;
    end process ctrl_unit;

end behave;
```

Annexe 16 -archi_i960_1_code.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use work.data_type.all;
use work.arch_type_i960_1_code.all;

entity archi_i960_1_code is
port(
            object_2_my_port_data_in : in PIC_int:=0;
            object_2_my_port_data_out : inout PIC_sum:=(0,0);
            object_2_my_port_start : out std_logic:='0';
            object_2_my_port_unlock : in std_logic;
            object_3_receptor_data_in : in PIC_sum:=(0,0);
            object_3_receptor_data_out : inout PIC_int:=0;
            object_3_receptor_start : in std_logic;
            object_3_receptor_unlock : out std_logic;
            cs_bar          : in std_logic;
            oe             : in std_logic;
            we           : in std_logic;
            data            : inout data_bus_i960_1_code;
            address         : in  address_bus_i960_1_code;
            w_rbar          : in std_logic;
            clk             : in std_logic;
            reset           : in std_logic
);
end archi_i960_1_code;

architecture picasso of archi_i960_1_code is

component architec_i960_1_code
port(
      cs_bar          : in std_logic;
      oe           : in std_logic;
      we           : in std_logic;
      data            : inout  data_bus_i960_1_code;
      copro_data_in   : out  data_bus_i960_1_code;
      copro_data_out  : in  data_bus_i960_1_code;
      address         : in  address_bus_i960_1_code;
      w_rbar          : in std_logic;
      clk             : in std_logic;
      reset           : in std_logic;
      copro_ask    : in copro_ask_type_i960_1_code;
      copro_ack    : out copro_ask_type_i960_1_code;
      resume_copro       : out copro_ask_type_i960_1_code;
      copro_mem_cs       : out std_logic
);
end component;

component copro_master
port(
```

```
      unlock              : out std_logic;
      clk              : in std_logic;
      reset            : in std_logic;
      copro_start  : in std_logic;
      copro_ask    : out std_logic;
      copro_ack    : in std_logic;
      resume_copro       : in std_logic
);
end component;

component copro_slave
port(
      unlock              : in std_logic;
      clk              : in std_logic;
      reset            : in std_logic;
      copro_start  : out std_logic;
      copro_ask    : out std_logic;
      copro_ack    : in std_logic;
      resume_copro       : in std_logic
);
end component;

component copro_type_i960_1_code
port(
      object_2_my_port_data_in : in PIC_int;
      object_2_my_port_data_out : inout PIC_sum;
      object_3_receptor_data_in : in PIC_sum;
      object_3_receptor_data_out : inout PIC_int;
      copro_mem_cs     : in std_logic;
      address          : in address_bus_i960_1_code;
      copro_data_in    : in data_bus_i960_1_code;
      copro_data_out   : out data_bus_i960_1_code;
      w_rbar       : in std_logic
);
end component;

signal copro_ask   : copro_ask_type_i960_1_code:=(others=>'0');
signal copro_ack   : copro_ask_type_i960_1_code:=(others=>'0');
signal resume_copro      : copro_ask_type_i960_1_code:=(others=>'0');
signal copro_mem_cs      : std_logic:='0';

signal copro_data_in,copro_data_out  : data_bus_i960_1_code;
begin
architec_bloc : architec_i960_1_code port map(
      cs_bar=>cs_bar,
      oe=>oe,
      we=>we,
      data=>data,
      copro_data_in=>copro_data_in,
      copro_data_out=>copro_data_out,
      address=>address,
      w_rbar=>w_rbar,
      clk=>clk,
      reset=>reset,
```

```
      copro_ask=>copro_ask,
      copro_ack=>copro_ack,
      resume_copro=>resume_copro,
      copro_mem_cs=>copro_mem_cs
);

slave_object_2_my_port:copro_slave port map(
      unlock=>object_2_my_port_unlock,
      clk=>clk,
      reset=>reset,
      copro_start=>object_2_my_port_start,
      copro_ask=>copro_ask(1),
      copro_ack=>copro_ack(1),
      resume_copro=>resume_copro(1)
);

master_object_3_receptor:copro_master port map(
      unlock=>object_3_receptor_unlock,
      clk=>clk,
      reset=>reset,
      copro_start=>object_3_receptor_start,
      copro_ask=>copro_ask(2),
      copro_ack=>copro_ack(2),
      resume_copro=>resume_copro(2)
);

mem_copro_type:copro_type_i960_1_code port map(
      object_2_my_port_data_in=>object_2_my_port_data_in,
      object_2_my_port_data_out=>object_2_my_port_data_out,
      object_3_receptor_data_in=>object_3_receptor_data_in,
      object_3_receptor_data_out=>object_3_receptor_data_out,
      copro_mem_cs=>copro_mem_cs,
      address=>address,
      copro_data_in=>copro_data_in,
      copro_data_out=>copro_data_out,
      w_rbar=>w_rbar
);

end picasso;
```

Annexe 17 -i_chip.vhd

```
-- This design instatntiates an i960ka bus interface and uses it to
-- access to local memory.
-- adapte pour un kx

library ieee;
use ieee.std_logic_1164.all;
use std.standard.all;

entity i_chip is
      port(
            W_R_bar          : inout std_logic :='1';
          archi_addr       :  out std_logic_vector(27 downto 0);
            LAD              : inout  std_logic_vector(31 downto 0);
          archi_cs_bar     :  inout std_logic;
            OE_bar           : inout std_logic;
            WE_bar           : inout std_logic;
          CLK              :  inout std_logic;
          RESET            : inout  std_logic
      );
end i_chip;

architecture picasso of i_chip is

------------------------------------------------------------------------
---
------------------------------------------------------------------------
---
--
-- COMPONENT DECLARATIONS
--
------------------------------------------------------------------------
---
------------------------------------------------------------------------
---

--i960kx Bus interface Model

component i960kx

port (
        LAD31_0                 : inout std_logic_vector(31 downto 0);
        LOCK_BAR                : inout std_logic;
        INT3_BAR_INTA_BAR                 : inout std_logic;
        CLK2                : in std_logic;
        READY_BAR               : in std_logic;
        HOLD                : in std_logic;
        BADAC_BAR               : in std_logic;
        RESET               : in std_logic;
        INT1                : in std_logic;
        INT2_INTR               : in std_logic;
        ALE_BAR               : out std_logic;
```

```
        ADS_BAR                  : out std_logic;
        W_R_BAR                   : out std_logic;
        DT_R_BAR                  : out std_logic;
        DEN_BAR                  : out std_logic;
        BE3_0_BAR                   : out std_logic_vector(3 downto 0);
        HLDA                : out std_logic;
        CACHE               : out std_logic;
        FAILURE_BAR             : out std_logic;
        INT0_BAR                : in std_logic
        );

end component;

for all :i960kx use entity work.i960kx(interface);

component address_decode is
    port(
         addr_in          : in   std_logic_vector( 31 downto 4 );
         bank_0_cs        : out  std_logic;
         bank_1_cs        : out  std_logic;
       picasso_io_cs     : out  std_logic;
       cs_fault    : out  std_logic
        );

end component;

for all :address_decode use entity work.address_decode(behave);

component address_latch is
    port(
         ale_bar          : in   std_logic;
         addr_in          : in   std_logic_vector( 31 downto 4 );
         addr_out         : out  std_logic_vector( 31 downto 4 )
        );

end component;

for all :address_latch use entity work.address_latch(behave);

component burst_logic is
    port(
         addr_in           : in   std_logic_vector( 3 downto 0 );
         den_bar           : in   std_logic;
         ads_bar           : in   std_logic;
         ready_bar         : in   std_logic;
         cycle_in_progress : out  std_logic;
         addr_3_2          : out  std_logic_vector( 3 downto 2 )
        );

end component;

component byte_enable_latch is
    port(
```

```
        ale_bar         : in   std_logic;  -- address latch enable
from cpu
        ready_bar       : in   std_logic;  -- ready from mem
controller
        be_in_bar       : in   std_logic_vector( 3 downto 0 ); -- in
from cpu
        be_out_bar      : out  std_logic_vector( 3 downto 0 )  -- out
to mem
       );

end component;

for all :byte_enable_latch use entity work.byte_enable_latch(behave);

component sram_if is
    port(
        oe_in_bar          : in   std_logic;
        we_in_bar          : in   std_logic;
        be_bar             : in   std_logic_vector( 3 downto 0 );
        oe_out_bar         : out  std_logic_vector( 3 downto 0 );
        we_out_bar         : out  std_logic_vector( 3 downto 0 )
       );

end component;

for all :sram_if use entity work.sram_if(behave);

component timing_control is
    port(
        sram_cs_bar        : in   std_logic;
        write_read_bar     : in   std_logic;
        clk2               : in   std_logic;
        den_bar            : in   std_logic;
        cycle_in_progress  : in   std_logic;
        ready_bar          : out  std_logic;
        sram_oe_bar        : out  std_logic;
        sram_we_bar        : out  std_logic
       );

end component;

for all :timing_control use entity work.timing_control(behave);

component sram is
   generic (NUM_DATA_BITS: INTEGER:=8;
            NUM_ADDR_BITS: INTEGER:=18);
   port (
        ADDR   : in std_logic_vector( NUM_ADDR_BITS-1 downto 0 );
        CE : in std_logic;
        DQ : inout std_logic_vector( NUM_DATA_BITS-1 downto 0 );
        OE : in std_logic;
        WE : in std_logic
       );
```

```
end component;


-------------------------------------------------------------------------
---
-------------------------------------------------------------------------
---
--
-- TOP LEVEL SIGNAL DECLARATION
--
-------------------------------------------------------------------------
---
-------------------------------------------------------------------------
---

signal LATCHED_ADDR     :  std_logic_vector(31 downto 0);
signal ALE_bar          :  std_logic :='Z';
signal ADS_bar          :  std_logic :='1';
signal R_W_bar          :  std_logic :='0';
signal DT_R_bar         :  std_logic :='1';
signal READY_bar        :  std_logic :='1';
signal LOCK_bar         :  std_logic :='1';
signal DEN_bar          :  std_logic :='1';
signal BE_bar           :  std_logic_vector(3 downto 0) := "1111";
signal HOLD             :  std_logic :='0';
signal HLDA             :  std_logic :='0';
signal CACHE            :  std_logic :='Z';
signal BADAC_bar        :  std_logic :='1';
--signal RESET_bar        :  std_logic :='0';
signal FAILURE_bar      :  std_logic :='1';
signal IAC_bar_INT0_bar :  std_logic :='1';
signal INT1             :  std_logic :='0';
signal INT2_INTR        :  std_logic :='0';
signal INT3_INTA_bar    :  std_logic :='1';


signal cycle_in_progress:  std_logic := '0';    -- burst and timing
control
--signal OE_bar           :  std_logic := '1';
--signal WE_bar           :  std_logic := '1';
signal SRAM_OE_bar      :  std_logic_vector(3 downto 0) := "1111";
signal SRAM_WE_bar      :  std_logic_vector(3 downto 0) := "1111";
signal LATCHED_BE_bar   :  std_logic_vector(3 downto 0) := "1111";


signal SRAM_READY_bar   :  std_logic := '1';
signal BANK_0_CS_bar    :  std_logic := '1';
signal BANK_1_CS_bar    :  std_logic := '1';
signal CS_FAULT         :  std_logic := '1';
signal ANY_CS_bar       :  std_logic := '1';
```

```
begin

addr_latch :  address_latch
    port map (
         ale_bar     => ALE_bar,
         addr_in     => LAD(31 downto 4),
         addr_out    => LATCHED_ADDR(31 downto 4)
        );

chip_select : address_decode
    port map(
         addr_in         => LATCHED_ADDR(31 downto 4),
         bank_0_cs       => BANK_0_CS_bar,
         bank_1_cs       => BANK_1_CS_bar,
       picasso_io_cs       => archi_cs_bar,
       cs_fault   => cs_fault
        );

burst_control :  burst_logic
    port map (
         addr_in             => LAD(3 downto 0),
         den_bar             => DEN_bar,
         ads_bar             => ADS_bar,
         ready_bar           => ready_bar,
         cycle_in_progress   => cycle_in_progress,
         addr_3_2            => LATCHED_ADDR(3 downto 2)
        );

be_latch :  byte_enable_latch
    port map (
         ale_bar     => ALE_bar,
         ready_bar   => READY_bar,
         be_in_bar   => BE_bar(3 downto 0),
         be_out_bar  => LATCHED_BE_bar(3 downto 0)
        );

readygen :  timing_control
    port map (
         sram_cs_bar         => ANY_CS_bar,
         write_read_bar      => W_R_bar,
         clk2                => CLK,
         den_bar             => DEN_bar,
         cycle_in_progress   => cycle_in_progress,
         ready_bar           => SRAM_READY_bar,
         sram_oe_bar         => OE_bar,
         sram_we_bar         => WE_bar
        );


sram_control :  sram_if
    port map (
         oe_in_bar   => OE_bar,
         we_in_bar   => WE_bar,
         be_bar      => LATCHED_BE_bar(3 downto 0),
```

```
          oe_out_bar => SRAM_OE_bar(3 downto 0),
          we_out_bar => SRAM_WE_bar(3 downto 0)
         );


cpu : i960kx
    port map (
        LAD31_0          => LAD,
        LOCK_BAR         => LOCK_BAR,
        INT3_BAR_INTA_BAR        => INT3_INTA_bar,
        CLK2         => CLK,
        READY_BAR        => READY_bar,
        HOLD         => HOLD,
        BADAC_BAR        => BADAC_bar,
        RESET        => RESET,
        INT1         => INT1,
        INT2_INTR         => INT2_INTR,
        ALE_BAR          => ALE_bar,
        ADS_BAR          => ADS_bar,
        W_R_BAR           => W_R_bar,
        DT_R_BAR          => DT_R_bar,
        DEN_BAR          => DEN_bar,
        BE3_0_BAR           => BE_bar,
        HLDA         => HLDA,
        CACHE        => CACHE,
        FAILURE_BAR      => FAILURE_bar,
        INT0_BAR         => IAC_bar_INT0_bar
        );

bank0_byte0 : sram
   port map (
         ADDR    => LATCHED_ADDR(19 downto 2),
         CE      => BANK_0_CS_bar,
         DQ      => LAD(7 downto 0),
         OE      => SRAM_OE_bar(0),
         WE      => SRAM_WE_bar(0)
        );
bank0_byte1 : sram
   port map (
         ADDR    => LATCHED_ADDR(19 downto 2),
         CE      => BANK_0_CS_bar,
         DQ      => LAD(15 downto 8),
         OE      => SRAM_OE_bar(1),
         WE      => SRAM_WE_bar(1)
        );
bank0_byte2 : sram
   port map (
         ADDR    => LATCHED_ADDR(19 downto 2),
         CE      => BANK_0_CS_bar,
         DQ      => LAD(23 downto 16),
         OE      => SRAM_OE_bar(2),
         WE      => SRAM_WE_bar(2)
        );
bank0_byte3 : sram
```

```
    port map (
          ADDR    => LATCHED_ADDR(19 downto 2),
          CE      => BANK_0_CS_bar,
          DQ      => LAD(31 downto 24),
          OE      => SRAM_OE_bar(3),
          WE      => SRAM_WE_bar(3)
         );


bank1_byte0 : sram
   port map (
          ADDR    => LATCHED_ADDR(19 downto 2),
          CE      => BANK_1_CS_bar,
          DQ      => LAD(7 downto 0),
          OE      => SRAM_OE_bar(0),
          WE      => SRAM_WE_bar(0)
         );
bank1_byte1 : sram
   port map (
          ADDR    => LATCHED_ADDR(19 downto 2),
          CE      => BANK_1_CS_bar,
          DQ      => LAD(15 downto 8),
          OE      => SRAM_OE_bar(1),
          WE      => SRAM_WE_bar(1)
         );
bank1_byte2 : sram
   port map (
          ADDR    => LATCHED_ADDR(19 downto 2),
          CE      => BANK_1_CS_bar,
          DQ      => LAD(23 downto 16),
          OE      => SRAM_OE_bar(2),
          WE      => SRAM_WE_bar(2)
         );
bank1_byte3 : sram
   port map (
          ADDR    => LATCHED_ADDR(19 downto 2),
          CE      => BANK_1_CS_bar,
          DQ      => LAD(31 downto 24),
          OE      => SRAM_OE_bar(3),
          WE      => SRAM_WE_bar(3)
         );


-- Clock generator

  r_w_bar <= not w_r_bar;

  ANY_CS_bar <= BANK_0_CS_bar and BANK_1_CS_bar and archi_cs_bar and
CS_FAULT;
  --RESET_bar <= not RESET;
  READY_bar <= SRAM_READY_bar;

  archi_addr<=latched_addr(29 downto 2);
process
```

```
begin
reset<='1';
wait for 150 ns;
reset<='0';
wait;
end process;

process
begin
clk<='0';
wait for 100 ns;
clk<='1';
wait for 100 ns;
end process;
end picasso;
```

Annexe 18 -client_v_final_vhdl.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_signed.ALL;
use work.data_type.all;
entity client_v_final_vhdl is
port(
add_data_in : in PIC_int:=0;
add_data_out : inout PIC_sum:=(0,0);
add_start : out std_logic:='0';
add_unlock : in std_logic;
clk : in  std_logic:='0';
reset : in  std_logic

);
end client_v_final_vhdl;
architecture picasso of client_v_final_vhdl is
begin
process
variable result:PIC_int;
begin
reset_loop: loop        --init
      add_data_out.x<=14;
      add_data_out.y<=17;
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;

--Start developped
while add_unlock='0' loop
    wait until clk'event and clk='1';
    if (reset='1') then exit reset_loop; end if;
end loop;
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;
add_start<='1';
while add_unlock='1' loop
    wait until clk'event and clk='1';
    if (reset='1') then exit reset_loop; end if;
end loop;
add_start<='0';
wait until clk'event and clk='1';
--end Start

      result:=(add_data_in);

      wait;

end loop reset_loop;
end process;
end picasso;
```

Annexe 19 -adder_v_final_vhdl.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_signed.ALL;
use work.data_type.all;
entity adder_v_final_vhdl is
port(
clk : in  std_logic;
reset : in  std_logic;
value_data_in : in PIC_sum:=(0,0);
value_data_out : inout PIC_int:=0;
value_start : in std_logic;
value_unlock : out std_logic

);
end adder_v_final_vhdl;
architecture picasso of adder_v_final_vhdl is
begin

value : process
variable i:PIC_int;
variable j:PIC_int;
variable k:PIC_int;
begin
reset_loop: loop        --init
value_unlock<='1';
while value_start/='1' loop
    wait until clk'event and clk='1';
    if (reset='1') then exit reset_loop; end if;
end loop;
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;
      i:=(value_data_in.x);
      j:=(value_data_in.y);
      value_data_out<=i+j;
value_unlock<='0';
while value_start='1' loop
wait until clk'event and clk='1';
if (reset='1') then exit reset_loop; end if;
end loop;
end loop reset_loop;
end process value;

end picasso;
```

Annexe 20 -clocker_final_vhdl.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_signed.ALL;
use work.data_type.all;

entity clocker_final_vhdl is
port(
clock : out  std_logic;
reset : out  std_logic

);
end clocker_final_vhdl;
architecture picasso of clocker_final_vhdl is
begin

process
begin
reset<='1';
wait for 150 ns;
reset<='0';
wait;
end process;

process
begin
clock<='0';
wait for 100 ns;
clock<='1';
wait for 100 ns;
end process;

end picasso;
```

Annexe 21 -picasso_top.vhd

```
--Generated by Picasso
--Copyright Yannick Heneault, 1999, 2000

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE ieee.std_logic_arith.all;
USE work.data_type.all;
USE work.arch_type_i960_1_code.all;

ENTITY picasso_top IS
END picasso_top;

ARCHITECTURE behave OF picasso_top IS

SIGNAL 10   :   address_bus_i960_1_code;
SIGNAL 11   :  std_logic;
SIGNAL 12   :  std_logic;
SIGNAL 13   :  data_bus_i960_1_code;
SIGNAL 14   :  PIC_int;
SIGNAL 15   :  PIC_sum;
SIGNAL 16   :  std_logic;
SIGNAL 17   :  std_logic;
SIGNAL 18   :  PIC_sum;
SIGNAL 19   :  PIC_int;
SIGNAL 110  :  std_logic;
SIGNAL 111  :  std_logic;
SIGNAL 112  :  std_logic;
SIGNAL 113  :  std_logic;
SIGNAL 114  :  std_logic;
SIGNAL 115  :  std_logic;
SIGNAL 116  :   std_logic;
SIGNAL 117  :   std_logic;

-- Component declaration
COMPONENT i_chip
      PORT (
            CLK          : inout  std_logic ;
            LAD          : inout   std_logic_vector(31 downto 0) ;
            OE_bar             : inout  std_logic ;
            RESET        : inout   std_logic ;
            WE_bar             : inout  std_logic ;
            W_R_bar            : inout  std_logic  ;
            archi_addr         : out  std_logic_vector(27 downto 0) ;
            archi_cs_bar             : inout  std_logic
      );
END COMPONENT;

COMPONENT client_v_final_vhdl
      PORT (
            add_data_in        : in  PIC_int ;
```

```
            add_data_out            : inout  PIC_sum ;
            add_start          : out  std_logic ;
            add_unlock         : in  std_logic ;
            clk          : in   std_logic ;
            reset        : in   std_logic
      );
END COMPONENT;

COMPONENT adder_v_final_vhdl
      PORT (
            clk          : in   std_logic ;
            reset        : in   std_logic ;
            value_data_in           : in  PIC_sum ;
            value_data_out          : inout  PIC_int ;
            value_start        : in  std_logic ;
            value_unlock            : out  std_logic
      );
END COMPONENT;

COMPONENT clocker_final_vhdl
      PORT (
            clock        : out   std_logic ;
            reset        : out   std_logic
      );
END COMPONENT;

COMPONENT archi_i960_1_code
      PORT (
            address            : in   address_bus_i960_1_code ;
            clk          : in  std_logic ;
            cs_bar             : in  std_logic ;
            data         : inout  data_bus_i960_1_code ;
            object_2_my_port_data_in            : in  PIC_int ;
            object_2_my_port_data_out           : inout  PIC_sum ;
            object_2_my_port_start        : out  std_logic ;
            object_2_my_port_unlock       : in  std_logic ;
            object_3_receptor_data_in           : in  PIC_sum ;
            object_3_receptor_data_out          : inout  PIC_int ;
            object_3_receptor_start       : in  std_logic ;
            object_3_receptor_unlock            : out  std_logic ;
            oe           : in  std_logic ;
            reset        : in  std_logic ;
            w_rbar             : in  std_logic ;
            we           : in  std_logic
      );
END COMPONENT;

BEGIN
-- Instance port mappings.
object_9 : archi_i960_1_code
            PORT MAP (
      address =>10,
      clk =>11,
      cs_bar =>12,
```

```
        data =>l3,
        object_2_my_port_data_in =>l4,
        object_2_my_port_data_out =>l5,
        object_2_my_port_start =>l6,
        object_2_my_port_unlock =>l7,
        object_3_receptor_data_in =>l8,
        object_3_receptor_data_out =>l9,
        object_3_receptor_start =>l10,
        object_3_receptor_unlock =>l11,
        oe =>l12,
        reset =>l13,
        w_rbar =>l14,
        we =>l15
        );

object_8 : i_chip
              PORT MAP (
        CLK =>l1,
        LAD =>l3,
        OE_bar =>l12,
        RESET =>l13,
        WE_bar =>l15,
        W_R_bar =>l14,
        archi_addr =>l0,
        archi_cs_bar =>l2
        );

object_7 : client_v_final_vhdl
              PORT MAP (
        add_data_in =>l9,
        add_data_out =>l8,
        add_start =>l10,
        add_unlock =>l11,
        clk =>l16,
        reset =>l17
        );

object_3 : adder_v_final_vhdl
              PORT MAP (
        clk =>l16,
        reset =>l17,
        value_data_in =>l5,
        value_data_out =>l4,
        value_start =>l6,
        value_unlock =>l7
        );

object_2 : clocker_final_vhdl
              PORT MAP (
        clock =>l16,
        reset =>l17
        );

END behave;
```

Annexe 22 -data_type.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

package data_type is

subtype PIC_int is integer;

subtype PIC_bool is std_logic;

type PIC_sum is record
      x:PIC_int;
      y:PIC_int;
end record;

end data_type;
```